

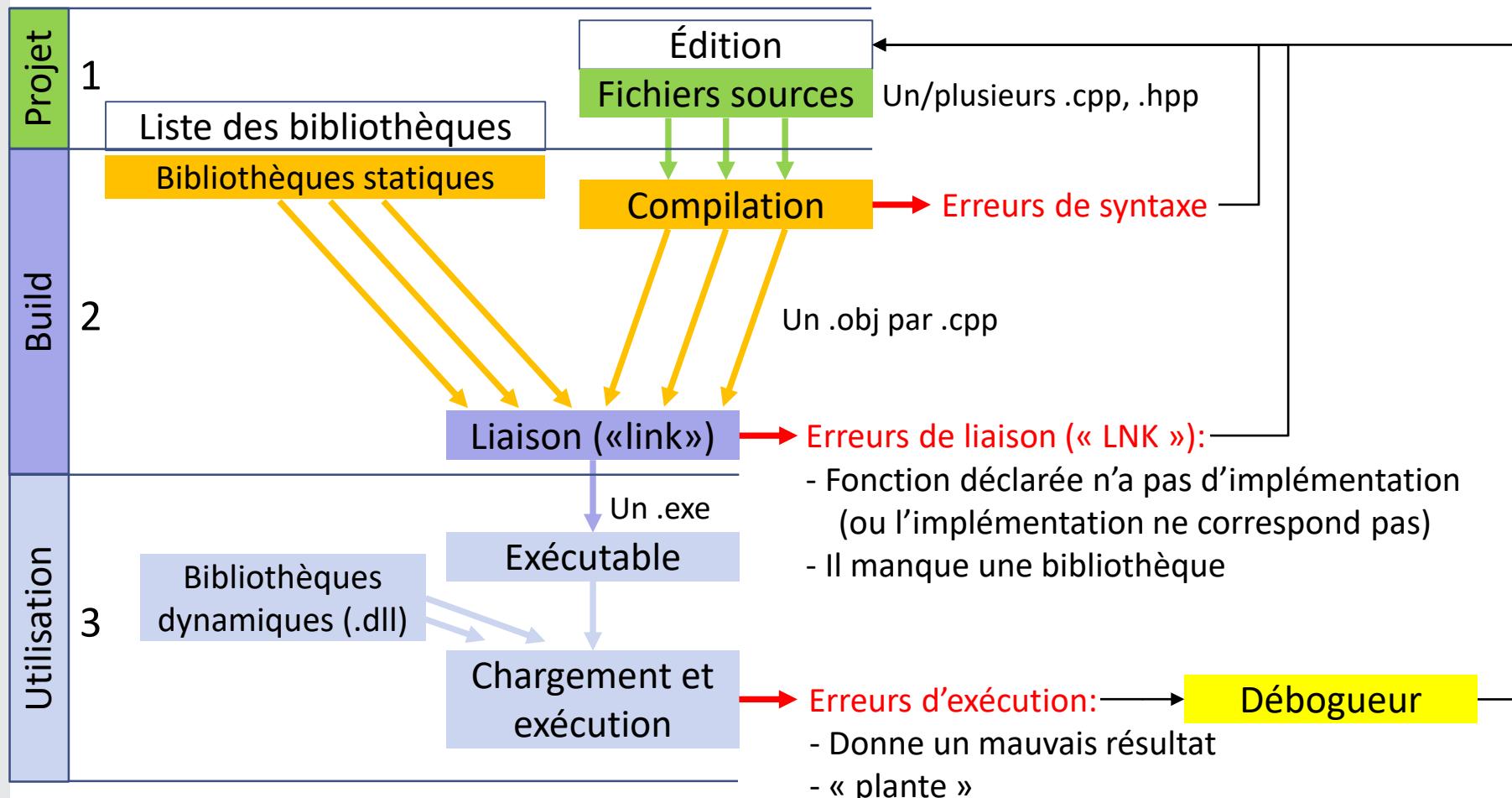
Introduction au C++

Concepts de base, variables, types

Pourquoi C++ ?

- Vitesse
 - Ex. $150 \times$ plus vite que Python: faire N fois: `a += valeur`
 - Car compilation → langage machine (INF1600), avec types statiques
- Contrôle plus fin de l'interaction avec le matériel
 - C inventé pour écrire Unix dans les années 1970 (d'où est basé MacOS et Linux)
- « Sur-ensemble » de C, qui permet un plus haut niveau d'abstraction
 - orienté objet, exceptions, templates, surcharge d'opérateurs, expressions lambda, coroutines (générateurs), ...
 - (pas parfaitement sur-ensemble, l'annexe C.1 d'ISO C++20 liste les quelques différences)
- C/C++ sont utilisés
 - C: noyaux de Windows, Linux, MacOS; cœur de Python;
 - C++: majorité des gros jeux (pas mobile); Windows (mais aussi C#).

Création d'un programme



Dans ce cours on utilise l'environnement Visual Studio pour toutes ces étapes.

Structure d'un programme

```
#include <iostream> //? Pour cin et cout.
```

Inclusions de bibliothèques ou nos propres déclarations

```
using namespace std;
```

Similaire à « from ... using * » en Python.

```
static const int factorielleMaxN = 12;
```

Déclarations/définitions

```
int factorielle(int n)
```

Définitions de fonctions ou méthodes

```
{  
    int produit = 1;  
    for (int i = 1; i <= n; ++i)  
        produit *= i;  
    return produit;  
}
```

Une fonction principale « main » (un seul des .cpp du projet contient *main*) de type *int* pour indiquer un numéro d'erreur

```
int main() //? Fonction principale, début du programme.
```

```
{
```

```
    cout << "Entrer un nombre inférieur à " << factorielleMaxN << " : " ;  
    int nombre; //? Déclaration d'une variable entière de nom "nombre".  
    cin >> nombre; //? Lecture du clavier dans la variable selon son type.
```

```
    cout << "La factorielle est: " << factorielle(nombre) << endl;
```

```
}
```

Si *main* n'a pas de *return*, donne 0 qui indique « pas d'erreur »

I_factorielle.cpp

Affichage / lecture du clavier

- *cout* et *cin* sont des variable « stream » vers l'écran et du clavier.
- *cout << a << b;* affiche à l'écran les valeurs de *a* et *b* sans fin de ligne ni espace ajouté.
- *cin >> a >> b;* lit du clavier et place la première valeur dans *a* et la suivante dans *b*
 - Valeurs séparées par espaces, tab ou enter
 - Lecture faite selon le type des variables
- *getline(cin, monTexte, caractèreFin);* pour lire un texte pouvant contenir des espaces et le mettre dans la variable *monTexte*
 - *caractèreFin* est par défaut '\n' (le caractère enter)

Identificateur

- Identificateur: nom d'une entité.
 - Nom d'une variable, constante, fonction...
- Composé de (comme en Python):
 - A-Z a-z 0-9 (caractères alphanumériques)
 - _ (soulignement)
 - ne débutant pas par un nombre
 - supporte les caractères accentués (depuis C++11)
- Les lettres minuscules et majuscules sont différencierées (comme en Python).
 - TP1, Tp1, tP1, tp1 sont quatre identificateurs distincts.

Les déclarations

Tout a un **type** qui doit être défini.

- constantes

```
const int tailleMax = 15;
```

- variables

```
int age = 10;
```

- définitions de types `typedef double Reel;`

```
struct Point { Reel x,y; };
```

- prototypes de sous-programmes

```
int sommer(int,int);
```

- macros

```
#define IFDEBUG(x) x
```

Instruction simple

Chaque ligne, terminée par un point-virgule :

```
int nbJours = 30;
```

Python a besoin de ; seulement pour plusieurs instructions sur une même ligne. C++ en a besoin à chaque instruction.

```
cout << "Donner deux valeurs entières ";
```

```
int nombre1, nombre2;
```

```
cin >> nombre1 >> nombre2;
```

```
cout << nbJours << nombre1 << nombre2;
```

I_instruction_simple.cpp

Instruction composée

Bloc d'instructions simples entre accolades.

Les variables d'un bloc n'existent plus après le bloc.

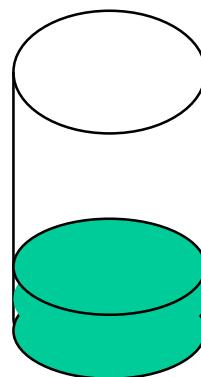
```
{  
    int numero = 12345;  
    string alliage = "aluminium";  
    double enStock = 3.45e12;  
    cout << numero << alliage << enStock;  
}  
// numero, alliage et enStock n'existent plus.  
{  
    cout << "Donner votre numéro de code";  
    int numCode;  
    cin >> numCode;  
    cout << numCode;  
}  
// numCode n'existe plus.
```

Pas comme Python.

I_instruction_composee.cpp

Type d'une variable

Une variable peut être comparée à ce verre. On peut ajouter et retirer du liquide selon sa bonne volonté.



Analogie:

| | |
|-------------|-------------------|
| Verre | Variable |
| - Contenant | - Adresse mémoire |
| - Contenu | - Valeur |

Le type indique :

- l'espace mémoire que le contenant prend,
- comment est mémorisé le contenu,
- les opérations pour le manipuler.

Types de base les plus utilisés

int Python est non borné

| Type | Description | Syntaxe | Valeurs* | Taille* |
|-----------------|--|--------------------------------|--|----------|
| <i>int</i> | nombre entier | 123 | ± 2 milliards | 4 octets |
| <i>unsigned</i> | nombre entier ≥ 0 ; utilisé pour les tailles | 123U <code>sizeof(x)</code> | 0 à 4 milliards | 4 octets |
| <i>double</i> | nombre réel | 123.0 6.02e23 | $\pm 10^{\pm 308}$ ~16 chiffres signif. | 8 octets |
| <i>char</i> | caractère ; élément de base pour les textes | 'A' (entre apostrophes) | 0-9 A-Z a-z ! " # \$ & ... | 1 octet |
| <i>bool</i> | booléen (vrai ou faux) ; pour les conditions | true false | true false | 1 octet |

Types les plus utilisés pour les textes

| | | |
|------------------------|--------------------------------|--|
| tableau de <i>char</i> | chaîne de caractères style C | "bonjour" |
| <i>string</i> | chaîne de caractères style C++ | <code>string("bonjour")</code> "bonjour"s (depuis C++14) |

Note*: Les valeurs limites et tailles peuvent varier selon le compilateur. Pour une configuration de projet, les tailles sont fixes.

Les entiers: int

Opérateurs arithmétiques

$+, -, *, /, \%$

Division entière comme // Python

Pas d'opérateur exposant, $**$ en Python.
Fonction $\text{pow}(a, b)$ pour a^b

Opérateurs $/$ et $\%$ $(a/b)*b + a\%b$ est égal à a

| Opérateur | Opération | Expression | Résultat |
|-----------|--|-------------------------------------|---------------------|
| $/$ | Quotient (division) (l'arrondi est vers zéro) | 11 / 4 -11 / 4 | 2 -2 (C++11) |
| $\%$ | Reste (« modulo ») (même signe que le numérateur) | 11 % 4 -11 % 4 | 3 -3 (C++11) |

En Python: $-11//4 = -3$ donc $-11\%4 = 1$

Opérateurs relationnels

$==, <, >, <=, >=, !=$

Comme Python

Les réels: double

Opérateurs arithmétiques

$+, -, *, /$

La division entre des opérandes réels donne un résultat réel

→ $5.0 / 2.0$ ou $5 / 2.0$ ou $5.0 / 2$ donne 2.5

5 / 2 donne quoi?

Opérateurs relationnels

$==, <, >, <=, >=, !=$

(attention que l'égalité est sensible aux imprécisions de calculs,
par exemple $1.0 / 49 * 49 \neq 1.0$)

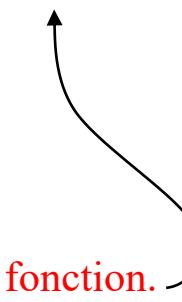
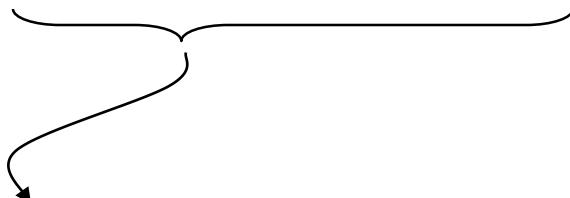
Comme Python

Fonctions sur les entiers et réels

#include <cmath>

⇒ Définies dans le fichier **cmath**

| Fonction | Description | Exemple |
|---|--|------------------------------|
| <code>double cos(double x)</code> | Cosinus de x (en radian) | $\cos(3.14) \approx -1.0$ |
| <code>double sin(double x)</code> | Sinus de x (en radian) | $\sin(1.57) \approx 1.0$ |
| <code>double exp(double x)</code> | e^x (e exposant x , où $e = 2,71828\dots$) | $\exp(2.0) \approx 7.389$ |
| <code>double pow(double x, double y)</code> | x^y (x exposant y) | $\text{pow}(3.0, 2.0) = 9.0$ |
| <code>double log(double x)</code> | Logarithme naturel de x ($\ln x$) | $\log(7.389) \approx 2.0$ |



Notation: *type du résultat fonction(type des paramètres)*

Les types ne doivent pas être écrits dans le programme lors de l'utilisation de la fonction.

Fonctions sur les entiers et réels

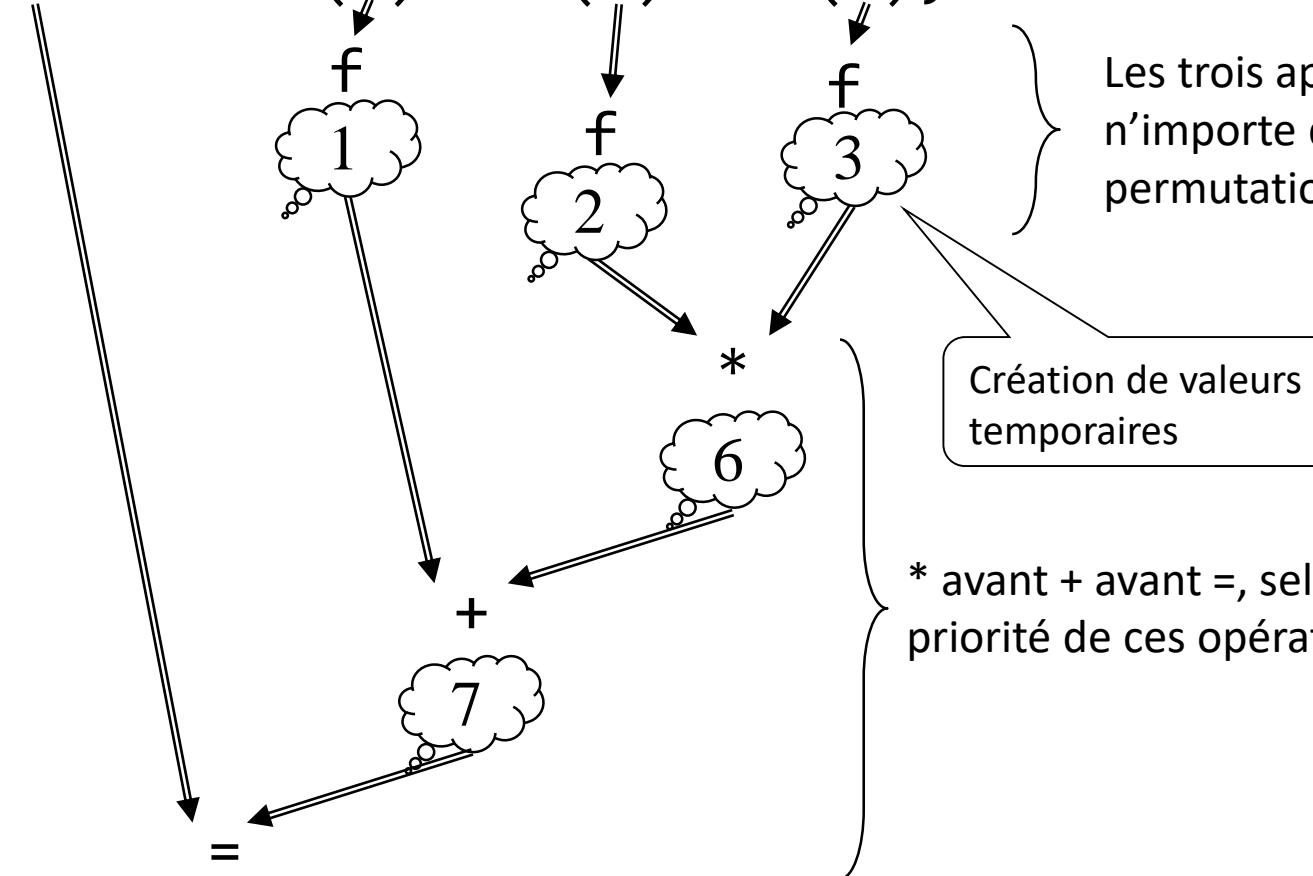
⇒ Définies dans le fichier **cmath**

| Fonction | Description | Exemple |
|-------------------------------------|---|---------------------------------|
| <code>double log10(double x)</code> | Logarithme en base 10 de x | <code>log10(100.0) = 2.0</code> |
| <code>double sqrt(double x)</code> | Racine carrée de x | <code>sqrt(16.0) = 4.0</code> |
| <code>int abs(int x)</code> | Valeur absolue de x (entier) | <code>abs(-2) = 2</code> |
| <code>double fabs(double x)</code> | Valeur absolue de x (réel) | <code>fabs(-12.3) = 12.3</code> |
| <code>double ceil(double x)</code> | $\lceil x \rceil$ Arrondi au plus petit entier $\geq x$ | <code>ceil(9.2) = 10.0</code> |
| <code>double floor(double x)</code> | $\lfloor x \rfloor$ Arrondi au plus grand entier $\leq x$ | <code>floor(9.8) = 9.0</code> |
| <code>double round(double x)</code> | Arrondi au plus proche entier | <code>floor(9.8) = 10.0</code> |

Priorité et ordre d'évaluation

Avec: `int f(int x) { cout << x; return x; }`

`resultat = f(1) + f(2) * f(3);`



Les trois appels sont dans n'importe quel ordre (6 permutations possibles).

Création de valeurs temporaires

* avant + avant =, selon la priorité de ces opérateurs.

`resultat` vaut 7, l'affichage est un de 123, 132, 213, 231, 312 ou 321 (les 6 permutations). Noter que certains opérateurs forcent un ordre d'évaluation.

Priorité des opérateurs et leur associativité

| Priorité | Opérateur | Associativité |
|----------|--|-----------------|
| 1 | <code>[] () (appel) -> . ++(postfixe)</code> <code>--(postfixe)</code> | gauche à droite |
| 2 | <code>! ~ ++(préfixe) --(préfixe)</code> <code>-(unaire) +(unaire) (type)</code> <code>*(unaire) &(unaire) sizeof</code> | droite à gauche |
| 3 | <code>* / %</code> | gauche à droite |
| 4 | <code>+ -</code> | gauche à droite |
| 5 | <code><< >></code> | gauche à droite |
| 6 | <code>< > <= >=</code> | gauche à droite |
| 7 | <code>== !=</code> | gauche à droite |
| 8 | <code>&</code> | gauche à droite |
| 9 | <code>^</code> | gauche à droite |
| 10 | <code> </code> | gauche à droite |
| 11 | <code>&&</code> | gauche à droite |
| 12 | <code> </code> | gauche à droite |
| 13 | <code>? : = += -= *= /= %=</code> <code>>>= <<= &= ^= =</code> | droite à gauche |
| 14 | <code>,</code> | gauche à droite |

En vert forcent un certain ordre
(généralement depuis C++17)

Opérateur , pas virgule entre paramètres.

++ et -- en position préfixe (avant) ou postfixe (après)

| Instruction | Même résultat que * |
|---------------------------------|---|
| <code>++i; ou i++;</code> | <code>i = i + 1;</code> |
| <code>--i; ou i--;</code> | <code>i = i - 1;</code> |
| <code>i = j++ + 4;</code> | <code>i = j + 4;</code> <code>j = j + 1;</code> |
| <code>i = ++j - 7;</code> | <code>j = j + 1;</code> <code>i = j - 7;</code> |
| <code>k = (--j) + (m++);</code> | <code>j = j - 1;</code> <code>k = j + m;</code> <code>m = m + 1;</code> |

Attention: ne pas écrire « i = i++; » ou « i = i--; »

* Ne pas utiliser la même variable deux fois dans une expression avec ++ ou --

Conversion implicite lors d'une opération

(pour les types vus dans ce chapitre)

| | <i>double</i> | <i>unsigned</i> | <i>int</i> | <i>char</i> | <i>bool</i> |
|-----------------|--|--|--|------------------------------------|--|
| <i>double</i> | $1.0 + 2.0 \rightarrow 3.0$ (symétrique...) | | | | <i>double</i> |
| <i>unsigned</i> | $1U + 2.0 \rightarrow 3.0$ | $1U + 2U \rightarrow 3U$ $1U - 2U \rightarrow \diamond$ | | | <i>unsigned</i> |
| <i>int</i> | $1 + 2.0 \rightarrow 3.0$ | $1 + 2U \rightarrow 3U$ $1 - 2U \rightarrow \diamond$ $-1 < 1U \rightarrow \text{false}$ | $1 + 2 \rightarrow 3$ $1 / 2 \rightarrow 0$ | | <i>int</i> |
| <i>char</i> | $'A' + 2.0 \rightarrow 67.0$ | $'A' + 2U \rightarrow 67U$ | $'A' + 2 \rightarrow 67$ | $'C' - 'A' \rightarrow 2$ | |
| <i>bool</i> | $\text{true} + 2.0 \rightarrow 3.0$ | $\text{true} + 2U \rightarrow 3U$ | $\text{true} + 2 \rightarrow 3$ | $\text{true} + 'A' \rightarrow 66$ | $\text{true} + \text{false} \rightarrow 1$ |

En rouge ci-dessus sont les résultats qui ne sont probablement pas ce qu'un humain s'attendrait; $\diamond = -1U = 4\,294\,967\,295$

```
// Affiche:    Type:
cout << 1 + true << endl;           // 2          int
cout << 1 + 0.5 << endl;            // 1.5        double
cout << 2.2 / 2 << endl;            // 1.1        double
cout << 1 / 2.0 << endl;            // 0.5        double
cout << -1 * 2 << endl;             // -2         int
// Ne donne pas ce qu'on pourrait croire :
cout << 1 / 2 << endl;              // 0          int
cout << 'A' + 2 << endl;            // 67         int, noter que 67=='C'
cout << -1 * 2U << endl;            // 4294967294 unsigned
```

II_conversion.cpp

Conversion explicite

Similaire à Python int(x) et float(x)

`type(x)` a comme résultat la conversion de la valeur de `x` en le type `type`. Aucune vérification faite.

Utiliser le moins possible.

- ⇒ `int('A' + 5.4)` L'addition donne un `double`, qui est ensuite converti en `int`.
- ⇒ `double(int(5.4) / 2)` La valeur 5.4 est convertie en `int`, la division entière est faite, le résultat est converti en `double`.

Autres façons (il y a différents types de ..._cast)

- `static_cast<int>(5.4 / 2);`
- `narrow_cast<int>(5.4 / 2);`
- `lround(5.4 / 2);`

De GSL; préférer si la perte de précision est voulue.

Une fonction qui arrondit avec résultat entier.

Conversion de type

```
// On aimeraient :
// 1 / 2 = 0.5 , 'A' + 2 = 'C' , -1 * 2U = -2 .
```

```
// Une des syntaxe C++ :
// Bonne manière :
```

```
cout << double(1) / 2 << endl; // 0.5 (division de double)
cout << char('A' + 2) << endl; // C (résultat converti en char)
cout << -1 * int(2U) << endl; // -2 (multiplication de int)
```

```
// Mauvaise manière :
```

```
cout << double(1 / 2) << endl; // 0 (division de int)
cout << 'A' + char(2) << endl; // 67 (résultat est un int)
cout << int(-1 * 2U) << endl; // -2 (bon résultat dans notre cas mais est
                                // "implementation-defined" selon C++11 §4.7¶3)
```

Si on a vraiment une constante, la bonne manière est 1.0/2 ou 1/2.0

```
// Mêmes conversion mais avec la syntaxe C :
```

```
cout << (double)1 / 2 << endl; // 0.5
cout << (char)('A' + 2) << endl; // C
cout << -1 * (int)2U << endl; // -2
```

```
// Mauvaise manière :
```

```
cout << (double)(1 / 2) << endl; // 0
cout << 'A' + (char)2 << endl; // 67
cout << (int)(-1 * 2U) << endl; // -2
```

Le type booléen en C/C++ : bool

- ❑ Pas de type booléen en C avant C99, utilisait int 0 et 1.
- ❑ Est considérée FAUSSE toute variable, constante ou expression qui vaut 0 et VRAIE toute variable, constante ou expression différente de 0.
 - Similaire à Python
 - En minuscules
- ❑ Type bool avec valeurs true et false
 - Comme Python

| Opération | Opérateur | Op. alt. |
|-----------------------------|-----------|----------|
| Négation | ! | not |
| Conjonction (ET) | && | and |
| Disjonction inclusive (OU) | | or |
| Disjonction exclusive (OUE) | != | |

Comme Python, évaluation court-circuit:
 false && f() ou true || f() n'appellent pas f

Peut avoir besoin #include <ciso646>
 sur anciens compilateurs.

Le type caractère : char

"A" (guillemets doubles)
n'est pas de type char

- Un caractère est identifié en le plaçant entre apostrophes:
 - `char lettre = 'A';`
- Un caractère est représenté par un entier, les opérateurs sur les entiers peuvent donc être utilisés; ex.: `'A'+1 == 'B'`
- Il est possible d'utiliser les opérateurs relationnels sur les caractères (`==`, `!=`, `<`, `>`, `<=`, `>=`). Noter que toutes les majuscules sont avant les minuscules (`'Z' < 'a'`).
- Les fonctions sur les `char` sont définies dans le fichier `<cctype>`
 - `isalpha(x)` `isdigit(x)` `is...` retournent `int != 0` si vrai
 - `tolower(x)` `toupper(x)` retournent la lettre dans un `int`
 - Besoin de changement de type, ex.: `cout << char(toupper('a'))`

Les chaînes de caractères en C++ : string

- Déclaration: *string laChaine;*
- Une chaîne de caractères C est identifiée en la plaçant entre guillemets (l'affectation à la *string* la rend C++).
 - `string laChaine = "Que la vie est belle! ";`

'un texte' (apostrophes)
n'est pas valide
- L'accès à un caractère de la chaîne comme en Python:
 - *laChaine[2]* est le 3^e caractère de la chaîne
 - Le 1^e caractère est à la position 0

Pas de *laChaine[0:4]* ni
laChaine[-1] de Python
- Est modifiable: `laChaine[4] = 'm';`
- Opérateurs relationnels se basant sur l'ordre alphabétique.
`(==, !=, <, >, <=, >=)`
- Le type **string** et ses fonctions sont dans le fichier `<string>`, c'est un type de la bibliothèque standard C++.

Fonctions sur les chaînes de caractères

Similaire à Python

⇒ Définies dans le fichier **string**

| Opérateurs | Description | Exemple |
|--|---|--|
| <code>dest = source;</code> | Copie la chaîne <i>source</i> dans la chaîne <i>destination</i> . La <i>source</i> peut aussi être une chaîne C. | <code>string chaine;</code> <code>chaine = "bonjour";</code> // chaine contient "bonjour" |
| <code>dest += source;</code> | Concatène la chaîne ou caractère <i>source</i> à la chaîne <i>destination</i> . La <i>source</i> peut aussi être une chaîne C. | // si chaine contient "bonjour" <code>chaine += " les ami";</code> <code>chaine += 's';</code> // chaine contient "bonjour les amis" |
| <code>dest = source1 + source2;</code> | Concaténation de deux chaînes, ou une chaîne et un caractère, le résultat est affecté à <i>dest</i> . | <code>chaine = "bonjour";</code> <code>chaine = chaine + " les ami"</code> <code>+ 's';</code> // chaine contient "bonjour les amis" |
| <code>chaine1 \leq chaine2</code> (où \leq est un des opérateurs relationnels) | Compare <i>chaine1</i> à <i>chaine2</i> selon l'ordre alphabétique. Attention: les minuscules et les majuscules sont distinctes. | // si chaine1 contient "bonjour" // et chaine2 contient "bonsoir" <code>chaine1 < chaine2</code> vaut true <code>chaine2 < chaine1</code> vaut false <code>chaine1 != chaine2</code> vaut true |

Il doit y avoir une 'string' C++ au moins d'un côté de + ou comparaison, l'autre peut être une chaîne C.

Fonctions sur les chaînes de caractères

⇒ Définies dans le fichier **string**

| Fonction | Description | Exemple |
|--|---|---|
| <code>chaine.size()</code> ou <code>chaine.length()</code> ou <code>size(chaine)</code> C++17 ou <code>ssize(chaine)</code> C++20 | <p>Retourne la longueur de la <i>chaine</i>.</p> <p>Attention: le type du résultat est <code>size_t</code>, qui est unsigned, sauf pour <code>ssize</code> qui est signé.</p> | <pre>chaine = "bonjour les amis"; cout << chaine.size(); // affiche 16</pre> |
| <code>chaine.substr(pos, long);</code> <code>chaine.substr(pos);</code> <div data-bbox="86 691 508 871" style="background-color: yellow; padding: 10px; border-radius: 10px;"> Similaire à Python <code>chaine[pos:pos+long]</code> et <code>chaine[pos:]</code> </div> | <p>Retourne la sous-chaîne de longueur <i>long</i> à partir de la position <i>pos</i>.</p> <p>Si la longueur n'est pas spécifiée, il prend de la position <i>pos</i> jusqu'à la fin de <i>chaine</i>.</p> | <pre>chaine = "bonjour les amis"; autre = chaine.substr(8, 3); // autre contient "les"</pre> |
| <code>chaine.find(autre)</code> | <p>Cherche la chaîne de caractères <i>autre</i> dans <i>chaine</i> et retourne sa position.</p> <p>Si <i>autre</i> n'est pas dans <i>chaine</i>, la valeur de retour est <code>chainenpos</code>.</p> | <pre>chaine = "Bon matin"; autre = "matin"; cout << chaine.find(autre); // affiche 4</pre> |
| <code>chaine.find(autre, position)</code> | <p>Cherche la chaîne de caractères <i>autre</i> dans <i>chaine</i> à partir de <i>position</i> et retourne la position d'<i>autre</i>.</p> | <pre>chaine = "bonjour les amis"; autre = " "; cout << chaine.find(autre, 8); // affiche 11</pre> |

Fonctions sur les chaînes de caractères

⇒ Définies dans le fichier **string**

| Fonction | Description | Exemple |
|---|---|--|
| <code>chaine.erase(pos, nbre);</code> | Enlève <i>nbre</i> caractères de la <i>chaine</i> , à partir de la position <i>pos</i> . | <code>chaine = "bonjour les amis"; chaine.erase(7, 4); // chaine contient "bonjour amis"</code> |
| <code>chaine.insert(pos, ajout);</code> | Insère la chaîne <i>ajout</i> dans <i>chaine</i> , à partir de la position <i>pos</i> . | <code>chaine = "la lo lu"; chaine.insert(3, "le li "); // chaine contient "la le li lo lu"</code> |
| <code>chaine.replace(pos, long, autre);</code> | Remplace <i>long</i> caractères dans <i>chaine</i> par la chaîne <i>autre</i> , à partir de la position <i>pos</i> ; <i>chaine</i> peut être augmentée ou diminuée. | <code>chaine = "milou et tintin"; chaine.replace(6, 2, "le chien de"); // chaine est "milou le chien de tintin"</code> |
| <code>chaine.resize(longueur);</code> <code>chaine.resize(longueur, c);</code> | Redimensionne la <i>chaine</i> à la <i>longueur</i> spécifiée. La <i>chaine</i> peut être tronquée ou augmentée (en ajoutant des copies du caractère <i>c</i> , qui par défaut est nul). | <code>chaine = "bon beigne"; chaine.resize(7); // chaine contient "bon bei"</code> |

i / o pour input / output
f pour file

Les fichiers : ifstream et ofstream

⇒ Définis dans le fichier **fstream**

- On peut lire/écrire dans des fichiers comme de *cin* et *cout*
- ifstream fichierEntrée("nom du fichier lu");
 - Comme *cin*: fichierEntrée >> a >> b; getline(fichierEntrée, c); ...
- ofstream fichierSortie("nom du fichier écrit");
 - Comme *cout*: fichierSortie << a << b; ...
- Les fichiers se ferment automatiquement quand variable détruite (à l'accolade })
 - Peut aussi fermer avec fichier.close();
 - Ouvrir avec fichier.open("nom fichier");
- fichier.exceptions(ios::failbit);
 - Active qu'un échec lancera une exception (message d'erreur)

pas besoin de « with »
comme Python

vector C++ est similaire à list Python,
attention list en C++ est très différent

Tableaux, array et vector

⇒ Définies dans le fichier **array** et **vector**
(Tableau est de base qui vient du C)

- Sont des structures contenant plusieurs éléments de même type, avec indices 0, 1, 2 ...
- Tableau et *array* de taille fixée à la compilation,
vector de taille variable.
- Déclaration:
 - *TypeDesElements nomVariable[dim1][dim2]...[dimN];*
 - *array<TypeDesElements, dim> nomVariable;*
 - *vector<TypeDesElements> nomVariable(dim);*
- Indexés avec []: nomVariable[i1][i2]...[iN]

Sauve un peu d'espace et temps

(dim) non requis,
vide par défaut.

Similaire à *list* Python

Exemple déclaration d'un tableau

```
// Accolades vides pour un tableau rempli de zéros:
```

```
int liste1[3] = { };
```

```
// array<int,3> liste1 = { };
```

```
// vector<int> liste1(3); //? pas de { }
```

```
for (int valeur : liste1)
```

```
    cout << valeur << ' ';
```

```
// Résultat: 0 0 0
```

Les trois fonctionnent dans cet exemple

```
// Accolades avec valeurs pour spécifier le contenu:
```

```
int liste2[3] = { 2, 7, 8 };
```

```
// array<int,3> liste2 = { 2, 7, 8 };
```

```
// vector<int> liste2 = { 2, 7, 8 };
```

```
for (int valeur : liste2)
```

```
    cout << valeur << ' ';
```

```
// Résultat: 2 7 8
```

Similaire à Python: for valeur in liste1:
mais doit indiquer un type pour valeur

On doit faire une boucle, pas
cout << liste1; ou cout << liste2;

Tableau 2D

```
int matrice[6][7];
array<array<int, 7>, 6> matrice;
vector<vector<int>> matrice = vector(6, vector<int>(7));
```

Tableau garantit que [1][0] est à l'adresse suivante de [0][6]

matrice[2][2]

Les trois fonctionnent dans cet exemple

Les cases d'une ligne sont à des adresses contigües

matrice[1][6]

matrice[5][0]

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |

[] ne vérifie pas les bornes, mais vérifiées en débogage dans VS pour array et vector.

Opérations sur tableaux, **array** et **vector**

- Il n'y a pas d'opération d'affichage ou de lecture pré définie pour ces structures.
- *array* et *vector* ont les opérateurs d'affectation et comparaisons lexicographiques:
 - = et ==, !=, <, >, <=, >=
- Aucune opération sur les tableaux complets
 - Doit définir nos propres fonctions élément par élément.

Traitement des éléments d'un tableau

```
double vecteurA[10], vecteurB[10];
```

```
//- Initialisation du vecteur -  
for (double& valeur : vecteurA)  
    valeur = 1.0; //? modifie car double&
```

& dit que c'est une référence et non une copie, l'affectation modifie l'original

```
//- Affectation d'un tableau à un autre -  
for (auto i : range(ssize(vecteurA)))  
// for (int i = 0; i < ssize(vecteurA); i++)  
    vecteurB[i] = vecteurA[i];
```

range de cppitertools similaire à Python

Ou ancien for de C/C++

I_tableau_vecteur.cpp

Traitement des éléments d'un tableau

```
//- Comparaison de deux tableaux -  
bool estPareil = true;          Peut mettre plusieurs conditions dans for  
for (int i = 0; i < ssize(vecteurA) && estPareil; i++)  
    if (vecteurA[i] != vecteurB[i])  
        estPareil = false;
```

```
//- Calculer la norme d'un vecteur -  
double norme = 0.0;  
for (double valeur : vecteurA)  
    norme += pow(valeur, 2);  
norme = sqrt(norme);
```

Tableau à 2 dimensions

```

// Initialisation d'un générateur aléatoire.
default_random_engine aleatoire(random_device{}());

const int dimension1 = 3, dimension2 = 5;
int matrice[dimension1][dimension2];

// Initialisation de la matrice avec valeurs aléatoires.
//? Pour passer chaque cases, 2 boucles imbriquées:
for (auto i : range(ssize(matrice)))           // ligne
    for (auto j : range(ssize(matrice[i])))       // colonne
        matrice[i][j] = aleatoire() % 1000;

// Afficher le contenu du tableau 2D.
for (const auto& ligne : matrice) {
    for (int valeur : ligne)
        cout << valeur << "\t" ;
    cout << endl;
}

```

Exemple avec range

Exemple « pour chaque ».
const: ne pas modifier.
auto&: référence vers une ligne du tableau (type avec syntaxe compliquée)

I_tableau_2D.cpp

Maximum d'un tableau à 2 dimensions

```
// Trouver la position i,j de la valeur maximum.  
int maximum = matrice[0][0];  
int positionIMax = 0, positionJMax = 0;  
for (int i = 0; i < ssize(matrice); i++) {  
    for (int j = 0; j < ssize(matrice[i]); j++) {  
        if (matrice[i][j] > maximum) {  
            // Conserver la valeur et indices i,j du maximum.  
            maximum = matrice[i][j];  
            positionIMax = i;  
            positionJMax = j;  
        }  
    }  
}  
cout << "maximum " << maximum << endl;  
cout << "indiceMax Ligne " << positionIMax << endl;  
cout << "indiceMax Colonne " << positionJMax << endl;
```

Exemple ancien for

Les enregistrements / structures

- *struct* définit un type formé de champs ayant des noms et pouvant être de types différents, comme une classe.
- Permet de regrouper des données diverses, mais logiquement interreliées, comme un objet.

```
struct NomDuType {  
    déclaration des champs;
```

```
};
```

; après l'accolade, car on peut déclarer une variable du type sans même donner de nom au type:
`struct { string rue; int numero; } maVariableAdresse;`

Les enregistrements / structures

```
struct Adresse { // Définition d'un type Adresse étant un enregistrement
    int numero;
    string rue, ville, codePostal;
};
```

N'importe quels types, incluant les tableaux et d'autres struct

```
struct Membre { // Définition d'un type Membre étant un enregistrement
```

```
    string nom, prenom;
    int age;
    char sexe;
    Adresse adresse;
    string telephone;
    double montantDu;
};
```

Le type Adresse déclaré ci-dessus

```
int main()
{
```

// Définition de variables de type Membre { valeurs des champs dans l'ordre }

```
Membre etudiant, clubSocial[50];
```

```
Adresse adresse = { 2900, "Edouard-Montpetit", "Montreal", "H3T1J4" };
```

```
Membre membre = { "Doe", "John", 18, 'M', adresse, "514-555-1234", 0.0 };
```

```
clubSocial[0] = membre;
```

Affectation copie tous les champs

```
cout << clubSocial[0].prenom;
```

Accès avec . comme objets Python

IV_enregistrements_membre.cpp

Les enregistrements / structures opérations

- L'affectation = copie tous les champs
- variable.champ pour accéder à un champ
- Aucune autre opération prédéfinie sur struct au complet, sauf C++20:

- Pour définir == et != qui compare tous les champs:
struct A {

```
    ...  
    bool operator==(const A&) const = default;  
};
```

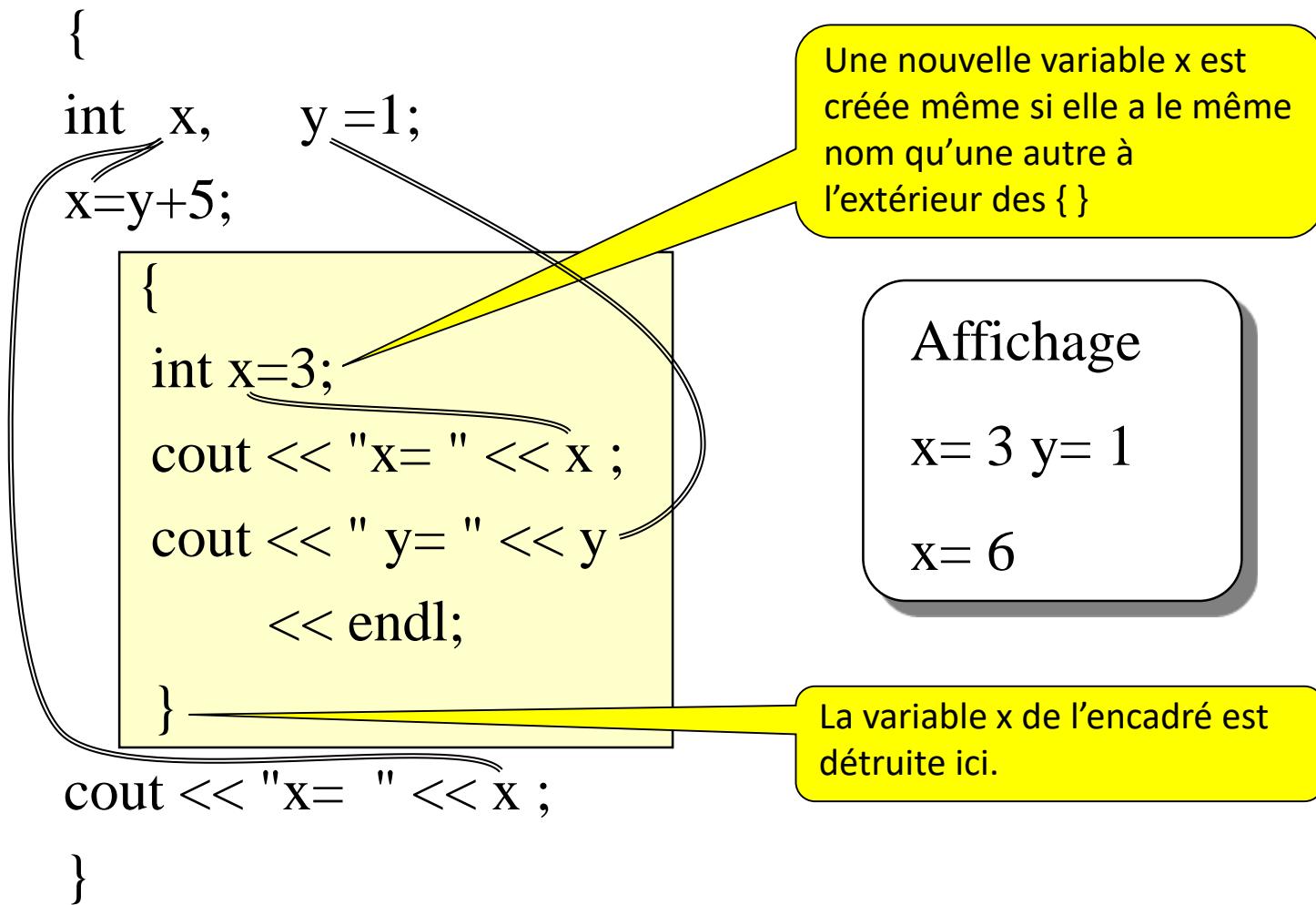
- Pour définir toutes les comparaisons, en ordre lexicographique selon l'ordre des champs:

```
struct A {  
    ...  
    auto operator<=>(const A&) const = default;  
};
```

Besoin de #include <compare>

Version actuelle de VS a besoin de *strong_ordering* au lieu *auto*, si la *struct* contient certains types comme *string*

Domaine de validité



Variables locales et globales

```
double taux;
```

Variable globale

```
double moyenne( double tableau[], int nombre)
{ int i;
  double moy;
..... // utilisation de la variable taux
}
int main()
{ double max, min, moy;
  double liste[20];
  int nombreElements;
..... // utilisation de la variable taux
  moy = moyenne(liste, nombreElements);
.....
}
```

Éviter d'utiliser les variables globales dans ce cours
(guide de codage point 48)

Pas de lien entre les deux variables locales *moy*

Même variable *taux* globale

Les constantes globales sont tout à fait permises.

« Variables constantes »

```
void main()
{
    static const double pi = 3.1416;
}
```

Constante locale

Les constantes « ne peuvent pas être modifiées » après leur déclaration.
Par contre:

```
void f(int x)
{
    const int y = x + 1;
    static const int z = x + 1;
    ...
}
```

‘y’ a une valeur différente à chaque appel

‘z’ a la même valeur pour tous les appels;
le premier appel lui fixe sa valeur

Constante à la compilation

Les variables les plus constantes sont ‘static const’; ‘static constexpr’ (C++11)
pour les types qui le supportent.

Introduction au C++

Structures de contrôle

Condition « if »

C++

```
if (a) {  
    b;  
}  
else if (c) {  
    d;  
}  
else {  
    e;  
}
```

Parenthèses autour de la condition

Accolades requises s'il y a plus d'une ligne d'instruction

; aux fins de lignes

« else if » au long

L'indentation n'est pas forcée par C++ mais est nécessaire pour la lisibilité (est notée dans le guide de codage)

Python

```
if a:  
    b  
elif c:  
    d  
else:  
    e
```

Condition « switch/case »

```
switch (expression) {  
    case constante_1 : instruction_1;  
        break;  
    case constante_2 :  
    case constante_3 : instruction_2_3;  
        break;  
    ...  
    case constante_x : instruction_x;  
        break;  
    ...  
    default : instruction;  
}
```

Pas d'équivalent direct Python

expression doit donner un type « entier »

Exécute du *case* où *expression* == *constante*, jusqu'à *break*; ou } qui ferme *switch*

Condition « switch/case » (suite)

```
int points = 0;  
char note;  
cout << "Note = "; cin >> note;  
switch (note) {  
    case 'A' : points +=4;  
    case 'B' : points +=3;  
    case 'C' : points +=2;  
        break;  
    case 'D' : points = 1;  
        break;  
    default : points = 0;  
}  
cout << " Cette note vaut " << points;
```

char est un entier, le code ASCII

*Quelle est la valeur de *points* si *note* = 'B' ?*

Boucle « while » et « do while »

C++

```
while (condition) {
```

```
    b;
```

```
}
```

```
do {  
    b;  
} while (condition);
```

Accolades requises s'il y a plus d'une ligne d'instruction

condition testée avant, donc *b* pas exécuté si déjà fausse avant première itération

Ne pas oublier « do »; syntaxiquement valide sans « do » mais ne fait pas la bonne chose

condition testée après une première exécution de *b*

Python

```
while condition:
```

```
    b
```

Pas d'équivalent direct

```
while True:
```

```
    b
```

```
if not condition:  
    break
```

Boucle « for »

C++

```
for (int i = 0; i < N; i++) { b; }
```

```
for (int i : range(N)) { b; }
```

```
for (int i : views::iota(0, N)) { b; }
```

```
for (type x : valeurs) { b; }
```

```
for (auto&& [i,x] : enumerate(valeurs)) {  
    b;  
}
```

Ancien *for* encore très commun
(en C depuis les débuts)

cppitertools

C++20
(pas encore
dans VS)

Python

```
for i in range(N): b
```

```
for x in valeurs: b
```

```
for i,x in \  
    enumerate(valeurs):  
        b
```

Doit être *auto* car ne peut
pas indiquer deux types
pour *i* et *x*

Boucle « for »

for (initialisation; expression_booléenne; actualisation)
instruction; 3

S'exécute 1, 2 (si vrai, continue), 3, 4, 2, 3, 4, ..., 2 (si faux, termine)

Équivalent à:

```
{  
    initialisation;  
    while (expression_booléenne) {  
        instruction;  
        actualisation;  
    }  
}
```

Sauf pour *continue* qui va à l'*actualisation* dans *for*

Une variable déclarée dans l'*initialisation* ou dans le *for* n'existe pas après le *for*

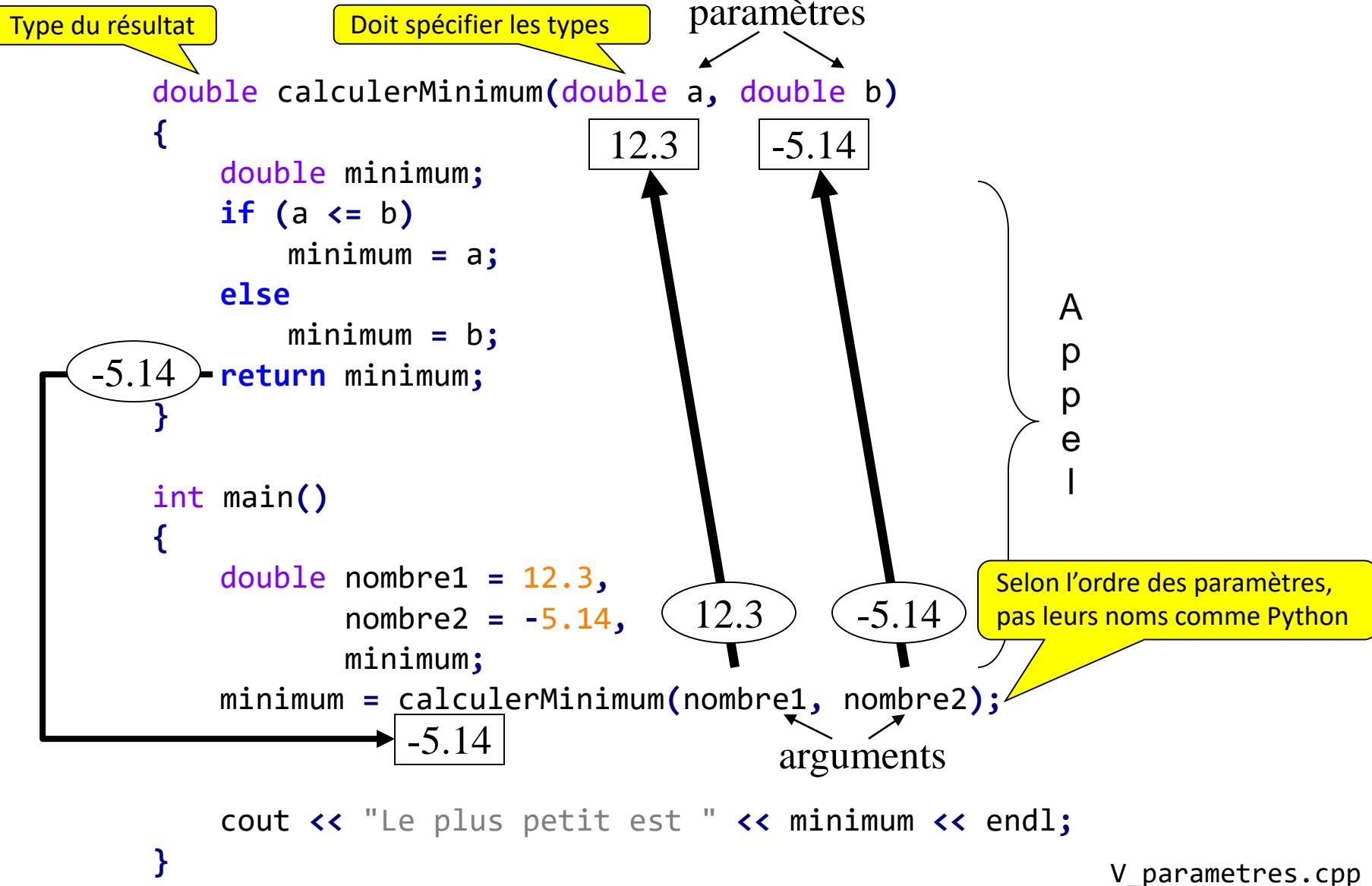
Pour lisibilité du *for*, *instruction* ne devrait normalement pas modifier la variable de *actualisation*

Exemple: for (int i = 0; i < 10; i++) cout << i; *i* n'existe plus après la boucle

Passage de paramètres

Paramètres, valeur de retour,
passage par valeur et adresse,
passage de tableaux et struct

Fonction: Les paramètres



Les mots « transmettre/transmission » et « passer/passage » sont utilisés pour les paramètres ; « passer » semble plus populaire.

Transmission de paramètres

Par valeur

- La valeur de l'argument est copiée dans le paramètre correspondant.
- Lorsque le paramètre est modifié, l'argument n'est pas modifié.

Par adresse

- L'adresse de l'argument est transmise au paramètre correspondant sans copie.
- Le paramètre réfère à la valeur de l'argument.
- Lorsque le paramètre est modifié, l'argument est modifié.

Transmission par valeur

void pour indiquer que la fonction ne retourne rien

```
void permuter(int a, int b) // Passage par valeur.
```

```
{
```

```
    int tampon = a;
```

```
    a = b;
```

```
    b = tampon;
```

```
}
```

```
int main()
```

```
{
```

```
    int i = 5;
```

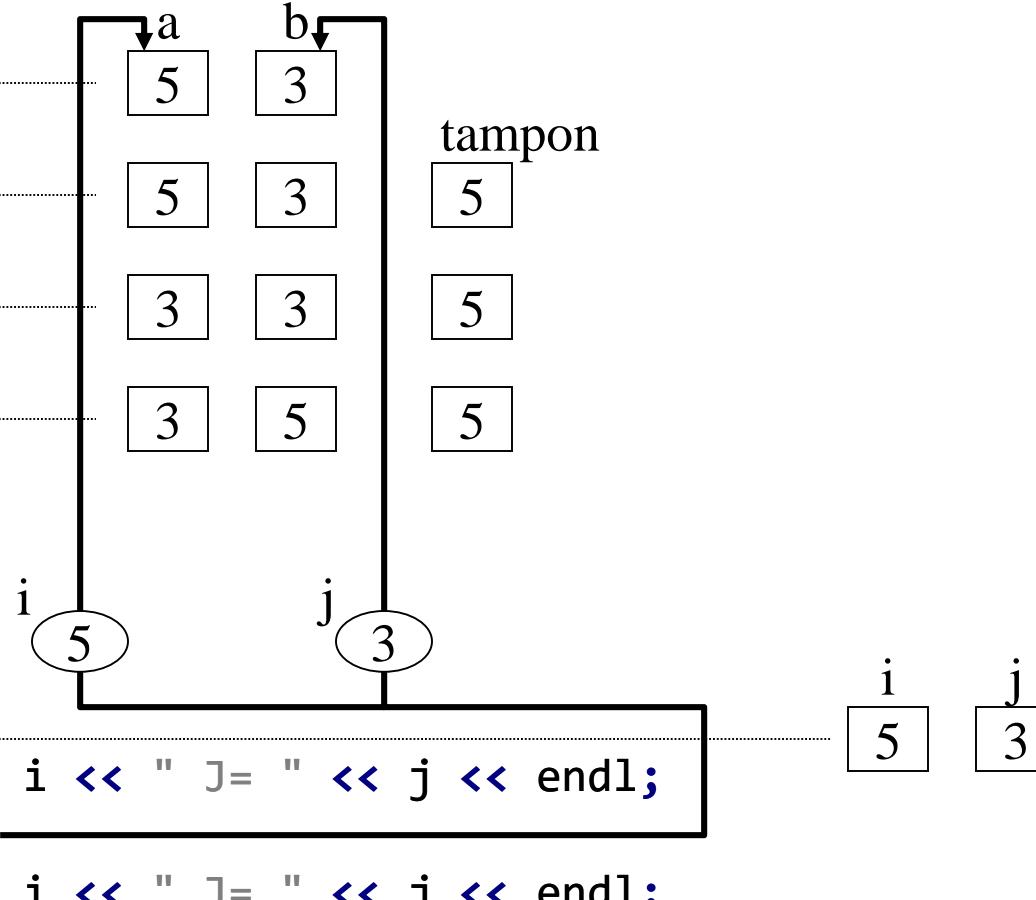
```
    int j = 3;
```

```
    cout << "I= " << i << " J= " << j << endl;
```

```
    permuter(i, j);
```

```
    cout << "I= " << i << " J= " << j << endl;
```

```
}
```



V_passage.cpp

Transmission par adresse

```

void permuter(int& a, int& b) // Passage par adresse.
{
    int tampon = a;
    a = b;
    b = tampon;
}

int main()
{
    int i = 5;
    int j = 3;
    cout << "I= " << i << " J= " << j << endl;
    permuter(i, j);
    cout << "I= " << i << " J= " << j << endl;
}

```

The diagram illustrates the state of memory during the execution of the program. It shows two main sections: the local scope of the `permuter` function and the global scope of the `main` function.

- Local Scope (`permuter`):** This section contains the code `int tampon = a;`, `a = b;`, and `b = tampon;`. It also shows the initial values of `i` and `j` being passed as arguments. The variable `tampon` is shown with its own pointer to the value 5. Arrows indicate the flow of pointers from `a` to `i` and `b` to `j`.
- Global Scope (`main`):** This section contains the initialization of `i` and `j`, followed by the output statements `cout << "I= " << i << " J= " << j << endl;` and the call to `permuter(i, j);`. After the call, the output statements show the swapped values: `I= 3 J= 5`.

V_passage.cpp

Transmission par valeur et par adresse

```
#include <iostream>
#include <iomanip>
using namespace std;

void modifier(int entier1, int& entier2)
{
    entier1 = entier1 + 100;
    entier2 = entier2 + 100;
    cout << "(2) Les valeurs sont ";
    cout << setw(4) << entier1 << setw(4) << entier2 << endl;
}

int main()
{
    int compteur = 7, indice = 12;

    cout << "(1) Les valeurs sont ";
    cout << setw(4) << compteur << setw(4) << indice << endl;

    modifier(compteur, indice);

    cout << "(3) Les valeurs sont ";
    cout << setw(4) << compteur << setw(4) << indice << endl;
}
```

Résultat de l'exécution:

```
(1) Les valeurs sont 7 12
(2) Les valeurs sont 107 112
(3) Les valeurs sont 7 112
```

V_passage_valeur_adresse.cpp

Cas standard

Doit bien documenter

Paramètres en entrée et de sortie

- Entrée (IN) : `void afficher(int x) { cout << x; }`
 - Le paramètre donne à la fonction une valeur dont elle a besoin.
 - On utilise le passage par valeur, si elle est facilement copiable, ou le passage par adresse *const* (ajouter *const* devant le type) pour éviter une copie mais ne pas modifier l'original.
- Sortie (OUT) : `void lireDans(int& x) { cin >> x; }`
 - Le paramètre donne à la fonction un endroit où placer un résultat.
 - On utilise le passage par adresse (non *const*) et la fonction y écrit le résultat.
 - La fonction ne lit pas ce qu'il y avait à cet endroit, il pouvait ne pas être initialisé avant l'appel.
- Entrée/sortie (IN/OUT) : `void incrémenter(int& x) { ++x; }`
 - Le paramètre donne à la fonction un endroit à modifier.
 - Similaire à (OUT) mais la fonction utilise la valeur avant d'y écrire une nouvelle; l'argument doit donc être bien initialisé.

Types des arguments

Pour exemple; on ne passe normalement pas *int* par *const int&*

- **void f(int x)** ou **void f(const int& x)**
 - L'argument peut être une expression d'un type implicitement convertible en le type du paramètre.
 - Ex: `f(1.2 + 4);` // 1.2 + 4 est le *double* 5.2, passé à *f* comme *int* 5
- **void f(int& x)** (x est OUT ou IN/OUT)
 - L'argument doit être un emplacement modifiable du type exact.
 - Ex: `int valeur = 4; f(valeur);`
 - Mauvais:
 - `double valeur = 4; f(valeur);` // mauvais type
 - `f(4);` // 4 est une constante (pas modifiable)
 - `int a = 1, b = 2; f(a + b);` // le résultat d'une expression (valeur temporaire) ne peut pas être passé comme référence non const.

Un tableau comme paramètre

- Toujours transmis par adresse (sans &) puisque l'affectation entre tableaux n'existe pas.
 - Un paramètre tableau est un pointeur vers le premier élément, il ne sait pas le nombre d'éléments; une variable tableau du bon type est passable à ce paramètre.
 - void modifier1D(int tableau[], int dim)
 - void modifier2D(int tableau[][], int dim)
 - void modifier3D(int tableau[][][], int dim)
 - void somme(int result[], const int a[], const int b[])

On utilise un autre paramètre pour indiquer la taille

Toutes les dimensions sauf la plus à gauche doivent être spécifiées

Paramètre OUT

Les paramètres IN devraient être *const* pour ne pas modifier

Exercice

- Écrire un programme qui calcule la note obtenue lors d'une compétition. Ce programme contient:
 - une fonction qui lit la note des 8 juges;
 - une fonction qui calcule la note totale en les additionnant sauf la note maximale et la note minimale;
 - une fonction qui trouve la note min et la note max.

Transmission des types composés (struct et types C++)

- Les types composés peuvent être transmis/retournés par valeur
 - Copie complète de tous les champs
 - Contrairement aux tableaux

```

par adresse                                par valeur
(4 ou 8 octets; 32 ou 64 bits)          (4408 octets + taille des textes)
↓                                         ↓
void tableauVsStruct(Ordinateur tableau[MAX_ORDIS], ListeOrdi structure)
{
    tableau[0].cpuGHz = 2.2;                // modifie le tableau original
    structure.liste[0].cpuGHz = 2.2; // modifie la copie de la structure
}

Avec: const int MAX_ORDIS = 50;           Taille possible: 2*32 +
struct Ordinateur {                      2*8 + 2*4 = 88
    string format, compagnie; double tailleEcranPouces, cpuGHz;
    int memoireRamGiB, memoireDisqueGB;
};

struct ListeOrdi { Ordinateur liste[MAX_ORDIS]; int nombreOrdi; };

```

Transmission des types composés (struct et types C++) (suite)

- Généralement transmis par adresse pour
 - Performance : le type peut être gros (**string**)
 - Simplicité : copier la valeur peut être problématique (...stream)
 - Ne nuit pas la lisibilité

```
void afficherOrdinateur(const Ordinateur& ordinateur)
{
    cout << ordinateur.compagnie << '\t'
        << ordinateur.format << '\t'
        << ordinateur.cpuGHz << '\t'
        << ordinateur.memoireRamGiB << '\t'
        << ordinateur.tailleEcranPouces << '\t'
        << ordinateur.memoireDisqueGB << endl;
}
```

- Attention de spécifier correctement le **const** si paramètre IN:
 - Soit l'expression: `afficher("bonjour");`
 - L'entête de fonction doit être: `void afficher(const string& texte);`

Transmission des types composés (struct et types C++) (suite)

- Les « stream » peuvent seulement être transmis par référence
 - Jamais const (ou presque): déplacer la tête de lecture est une modification

```
& pas const           paramètre OUT
```

```
void lireOrdinateur1(istream& fichier, Ordinateur& ordinateur)
{
    fichier >> ordinateur.format
            >> ordinateur.compagnie
            >> ordinateur.tailleEcranPouces
            >> ordinateur.cpuGHz
            >> ordinateur.memoireRamGiB
            >> ordinateur.memoireDisqueGB;
}
```

Transmission des types composés (struct et types C++) (suite)

- Le retour par valeur peut être optimisé par le compilateur
- Règle générale: écrire ce qui est le plus lisible
 - Si la performance est cruciale (jamais dans ce cours) vérifier si l'optimisation est faite

Probable copie de valeur, mais généralement plus lisible qu'un paramètre OUT



```
Ordinateur lireOrdinateur2(istream& fichier)
{
    Ordinateur ordinateur;
    fichier >> ordinateur.format
        >> ordinateur.compagnie
        >> ordinateur.tailleEcranPouces
        >> ordinateur.cpuGHz
        >> ordinateur.memoireRamGiB
        >> ordinateur.memoireDisqueGB;
    return ordinateur;
}
```

v_structTab.cpp

Transmission de tableaux

- Soit les variables:

```
int vals[] = { 1, 5, 3 };
int nVals = int(size(vals)); // PAS 'sizeof' (nombre d'octets).
```

- Comment passer ce tableau à une fonction?
- Passage d'un tableau (plus tôt dans ce chapitre):

```
void afficher(const int tableau[], int dim) {
    for (int i = 0; i < dim; i++)
        cout << tableau[i] << " ";
```

- Doit passer 2 paramètres pour un seul tableau
- Lien entre tableau et dim pas très clair (inconnu par C++)
- Appel:
`afficher(vals, nVals);`
- Comment lier clairement le tableau et sa taille?

V_passage_tableau.cpp

Transmission de tableaux (suite)

- Utilisation d'une struct:

```
struct TableauInt {  
    const int* valeurs; // Équivalent du paramètre const int tableau[]  
    int nValeurs; // Équivalent du paramètre int dim  
};
```

```
void afficher_v2(TableauInt tableau) {  
    for (int i = 0; i < tableau.nValeurs; i++)  
        cout << tableau.valeurs[i] << " ";
```

- Lien évident entre le tableau et sa taille
- Appels:

```
afficher_v2({vals, nVals}); // Accolades pour construire la struct.  
TableauInt tab = { vals, nVals };  
afficher_v2(tab); // Passage d'une struct existante.
```

- Mais

- Doit passer la bonne taille
- Doit définir une struct par type

- Peut-on faire mieux?

V_passage_tableau.cpp

Transmission de tableaux en C++20

⇒ Définis dans le fichier **span**

- Type « span » : généralisation de la struct précédente

```
void afficher_v3(span<const int> tableau) {
    for (int i = 0; i < tableau.size(); i++) // .size() comme pour string.
        cout << tableau[i] << " ";           // Accès comme un tableau.
```

- Appels:

```
afficher_v3(vals);                  // Conversion en span implicite.
afficher_v3({vals, nVals});        // Peut construire span comme la struct.
span s = vals;
afficher_v3(s.subspan(1, 2));       // Prend une partie d'un span.
                                    // .subspan(index_début, nombre_éléments)
afficher(s.data(), s.size());      // Pour utiliser l'ancienne fonction.
```

- Syntaxe dite *template* : le type des éléments entre < >
- Pas encore de taille variable multiples dimensions
 - `span<int[10]> tableau` fonctionne pour l'équivalent de `int tableau[][10]`

Emplacement des fonctions

```

#include <iostream>
#include <iomanip>
using namespace std;

// Computes the total cost, including 5% sales tax,
// on nItems items at a cost of itemPrice each.
double totalCost(int nItems, double itemPrice); ← Déclaration de fonction
ou prototype
(doit être avant l'utilisation)

int main()
{
    double pricePerItem;
    int numberOfItems;
    cout << "Enter the number of items purchased: ";
    cin >> numberOfItems;
    cout << "Enter the price per item $";
    cin >> pricePerItem;

    double bill = totalCost(numberOfItems, pricePerItem); ← Appel de fonction

    cout << fixed << showpoint << setprecision(2);
    cout << numberOfItems << " items at "
        << "$" << pricePerItem << " each." << endl
        << "Final bill, including tax, is $" << bill
        << endl;
}

double totalCost(int nItems, double itemPrice) ← Entête de fonction
{
    static const double TAXRATE = 0.05; // 5% sales tax
    double subtotal = itemPrice * nItems;
    return subtotal + subtotal*TAXRATE; } Corps de la fonction } Définition de la fonction
(peut être après l'utilisation)

```

V_emplacement_apres.cpp

```

#include <iostream>
#include <iomanip>
using namespace std;

// Computes the total cost, including 5% sales tax,
// on nItems items at a cost of itemPrice each.
double totalCost(int nItems, double itemPrice)
{
    static const double TAXRATE = 0.05; // 5% sales tax
    double subtotal = itemPrice * nItems;
    return subtotal + subtotal*TAXRATE;
}

int main()
{
    double pricePerItem;
    int numberOfItems;
    cout << "Enter the number of items purchased: ";
    cin >> numberOfItems;
    cout << "Enter the price per item $";
    cin >> pricePerItem;

    double bill = totalCost(numberOfItems, pricePerItem); ← Appel de fonction

    cout << fixed << showpoint << setprecision(2);
    cout << numberOfItems << " items at "
        << "$" << pricePerItem << " each." << endl
        << "Final bill, including tax, is $" << bill
        << endl;
}

```

Entête de fonction

Corps de la fonction

Définition de la fonction
(doit être avant l'utilisation si pas de prototype)

Appel de fonction

V_emplacement_avant.cpp

Pointeurs et références

Pointeurs, références,
allocation dynamique,
pointeurs intelligents

Le type pointeur

- Une variable pointeur correspond à une variable qui contient l'adresse mémoire d'une donnée.
- Déclaration:
 - `int* ptrEntier;`
- L'opérateur unaire & «adresse de» est utilisé pour connaître l'adresse d'une donnée.
 - `int entier;`
 - `int* ptrEntier = &entier;`

étoile (*) vs éperluète (&)

(Note: « perluète » est le terme privilégié par l'Office québécois de la langue française, mais les termes « et commercial », « esperluette », « esperluète », et « éperluète » sont aussi communs)

Leur signification diffère selon leur emplacement:

Dans une déclaration: (i.e. « int* a », « int& a »)

- * définit un pointeur;
- & définit une référence à une variable, c'est-à-dire un nouveau nom permettant d'accéder à cette même variable; la référence doit être déclarée en paramètre ou être **absolument initialisée à la déclaration**

Dans les instructions, en opérateur unaire: (i.e. « *a », « &a »)

- * représente la déréférence, soit « le contenu à l'adresse pointée »
- & représente « l'adresse de »

Dans les instructions, en opérateur binaire: (i.e. « a * b », « a & b »)

- ce sont les opérations de multiplication (*) et de ET binaire (&).

Adresse et déréférence

| Adresse | Type | Nom | Valeur |
|---------|-------|-----|--------|
| 1000 | int | a | 1008 |
| 1004 | int* | b | 1000 |
| 1008 | int | c | 123 |
| 1012 | int** | d | 1004 |

Compilé en 32bits pointeur
et int font 4 octets

- cout << a; // Affiche quoi ? Quel est le type ?
- cout << b;
- cout << *b;
- cout << &d;
- cout << *d;
- cout << **d;
- cout << *a;
- cout << *(int*)a;

Noter que par défaut les *int* sont affichés en décimal et les pointeurs en hexadécimal, on ne demande pas cette distinction ici

Adresse et déréférence (suite)

| Adresse | Type | Nom | Valeur |
|---------|-------|-----|--------|
| ? | int | a | 1008 |
| ? | int* | b | • |
| ? | int | c | 123 |
| ? | int** | d | • |

- Généralement les adresses ne sont pas importantes.
- On représente alors les pointeurs par des flèches sans indiquer de valeur.
- `cout << a; cout << **d;`
- `*b = 42;`
- `cout << a;`
- Que vaut b ? Sa valeur a-t-elle changée dans l'affectation?

Allocation dynamique

- Permet d'allouer de l'espace à une variable lors de l'exécution et de libérer cet espace lorsque la variable devient inutile.
 - Allocation dynamique s'effectue à l'aide de l'opérateur new qui exige le type de la variable dynamique.
 - varPointeur = **new** type_donnée_pointée;
 - ou varPointeur = **new** type_données_pointées[n];
 - Exemple:
 - **int*** ptrEntier = **new int**;
- int* ptrEntier •————→ int (sans nom) (valeur non-initialisée)
- Peut être la valeur d'une variable
- Mémoire réservée / allouée

Allocation dynamique (suite)

- La libération de l'espace mémoire alloué dynamiquement s'effectue à l'aide de l'opérateur *delete* qui exige de préciser le pointeur correspondant
 - **delete** varPointeur;
 - ou **delete[]** varPointeur;
- Chaque *new* doit avoir un *delete* qui correspond
 - Généralement pas dans la même fonction.
 - Si on oublie un *delete* la mémoire est perdue, dit « fuite de mémoire » (« memory leak »).
- Exemple:

• **delete** ptrEntier;

int* ptrEntier •

int (sans nom)

(valeur indéterminée)

Mémoire déréservée / désallouée

Le pointeur existe toujours et n'a pas changé de valeur

Tableau dynamique

```
// Déclaration d'un tableau d'entiers à une dimension.  
int* vecteur;  
  
// Attribution d'un tableau de 150 entiers.  
vecteur = new int[150];  
  
// Adresses du tableau et des cases.  
cout << "Adresse du tableau: " << vecteur << endl;  
for (int i=0; i<3; i++)  
    cout << "&vecteur[" << i << "] est " << &vecteur[i] << ";\n";  
cout << endl;  
  
// Initialise chaque valeur du tableau à 0.  
for (int i=0; i<150; i++)  
    vecteur[i] = 0;  
  
// Remet en disponibilité l'espace mémoire du tableau d'entiers.  
delete[] vecteur;  
vecteur = 0;
```

Pointeur 0, *nullptr* (C++11), pointe vers rien. Souvent représenté sur schémas par une mise à la terre.

// Affiche:
// Adresse du tableau: 00032F00
// &vecteur[0] est 00032F00; &vecteur[1] est 00032F04; &vecteur[2] est
00032F08;

VII_tableau_dynamique.cpp

Tableau dynamique 2D

```
// Déclaration d'un tableau de réels à deux dimensions,  
// permettant des lignes de longueurs différentes:  
double** matrice;  
  
// Attribution de l'espace pour un tableau de 20 lignes x 30 colonnes:  
matrice = new double* [20]; // Espace pour les 20 pointeurs de lignes.  
for (int ligne=0; ligne<20; ligne++) // Pour chaque ligne,  
    matrice[ligne] = new double[30]; // espace pour 30 colonnes de la ligne.  
  
// Initialise chaque valeur de la matrice à zéro:  
for (int ligne=0; ligne<20; ligne++)  
    for (int colonne=0; colonne<30; colonne++)  
        matrice[ligne][colonne] = 0.0;  
  
// Libération de l'espace mémoire:  
for (int ligne=0; ligne<20; ligne++) // Pour chaque ligne,  
    delete[] matrice[ligne]; // libérer l'espace pour la ligne.  
  
delete[] matrice; // Libérer l'espace mémoire des pointeurs de lignes.  
matrice = 0;
```

Tableau dynamique 2D (suite)

```
// Déclaration d'un tableau de réels à "deux dimensions",
// en utilisant un tableau une dimension:
double* matrice2;

// Attribution de l'espace pour un tableau de 20 lignes x 30 colonnes:
int largeurMatrice2 = 30;
int hauteurMatrice2 = 20;
matrice2 = new double [hauteurMatrice2 * largeurMatrice2];

// Initialise chaque valeur de la matrice à zéro:
for (int ligne=0; ligne<20; ligne++)
    for (int colonne=0; colonne<30; colonne++)
        matrice2[ligne*largeurMatrice2 + colonne] = 0.0;

// Libération de l'espace mémoire:
delete[] matrice2;
```

New et delete et enregistrement

```
struct Date {  
    int mois;  
    int jour;  
    int annee;  
};  
  
int main()  
{  
    Date* ptrDate;  
    ptrDate = new Date;  
    ptrDate->mois = 10;  
    ptrDate->jour = 18;  
    ptrDate->annee = 1938;  
    cout << ptrDate->mois << "/" << ptrDate->jour << "/"  
        << ptrDate->annee << endl;  
    delete ptrDate;  
    ptrDate = 0;  
}
```

ptrEnregistrement->champ
est équivalent à
(*ptrEnregistrement).champ
et permet d'accéder au champ de
l'enregistrement pointé.

Références

- Une référence (& dans le type) réfère à un emplacement, similaire à un pointeur, mais...
- Toujours liée à l'emplacement lors de sa création, ne peut pas être reliée à un autre après
 - Dans un appel de fonction, la création est lors de chaque appel
 - Dans une boucle, la création est à chaque itération
- Ne peut pas être liée à `*nullptr`
- Toute opération sur la référence a le même effet que l'opération sur l'emplacement référencé
 - Incluant l'affectation
- Ex:
 - `int a = 4; int& b = a;`
 - `b = 7;`
 - `cout << a << boolalpha << (&a == &b); // affiche: 7true`

b réfère à a

Compare les adresses

Affectation à une référence C++ vs Python

C++

```
vector<int>          v1 = { 1, 4 };
vector<vector<int>> v2 = {{1}, {4}};
```

```
for (auto x1 : v1) x1 += 1;
for (auto x2 : v2) x2[0] += 1;
afficher(v1); // -> [1, 4]
afficher(v2); // -> [[1], [4]]
```

```
for (auto& x3 : v1) x3 += 1;
for (auto& x4 : v2) x4[0] += 1;
afficher(v1); // -> [2, 5]
afficher(v2); // -> [[2], [5]]
```

Sans & x prend une copie de valeur;
on modifie la copie pas l'original.

Avec & x réfère vers la valeur;
l'affectation change la valeur référencée.

Python

```
v1 = [1, 4]
v2 = [[1], [4]];
```

```
for x1 in v1: x1 += 1
for x2 in v2: x2[0] += 1
print(v1) # -> [1, 4]
print(v2) # -> [[2], [5]]
```

Tout est des références.

x réfère vers la valeur;
l'affectation relie la référence au même
endroit que l'autre référence.

Affectation à une référence C++ vs Python

(suite) - première itération des boucles

C++ x1 prend une copie de v1[0]
 x2 prend une copie de v2[0]
 x3 réfère à v1[0]
 x4 réfère à v2[0]

avant / après +=1

Sans &:

v1 = [1 , 4]

x1 = 12

v2 = [[1], [4]]

x2 =[12]

L'affectation modifie la valeur référencée

Avec &:

v1 = [12 , 4]

x3 =>

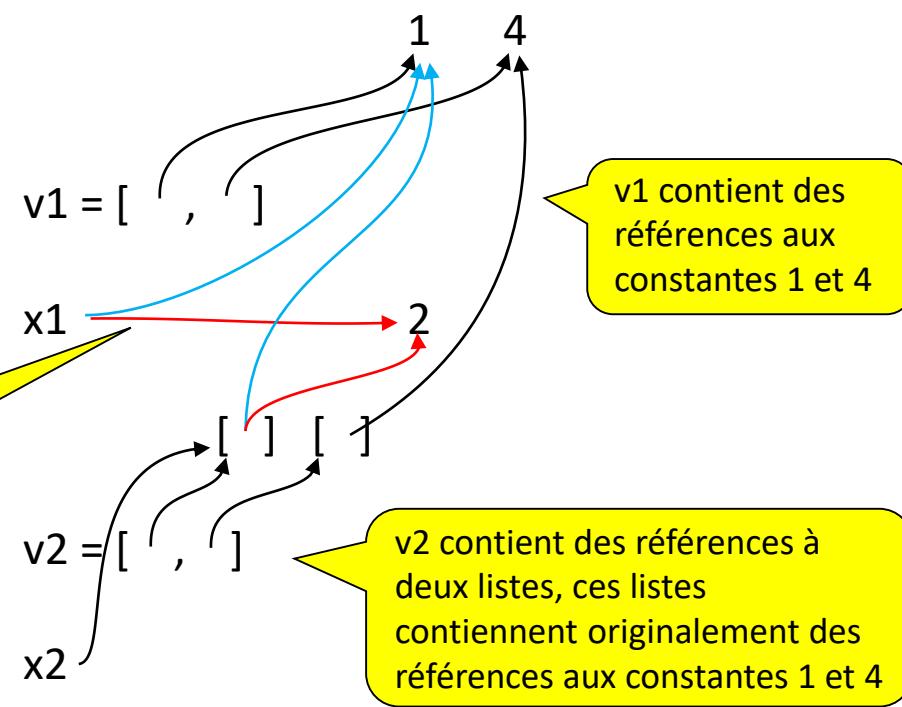
v2 = [[12], [4]]

x4 =>

L'affectation modifie où réfère la référence; possible en C++ avec des pointeurs

Python x1 réfère où réfère v1[0]
 x2 réfère où réfère v2[0]

avant / après +=1



Classes en C++

class, public, private,
constructeurs/destructeur,
méthodes const

Définition d'une classe en C++

```
class NomDeLaClasse {
```

```
public:
```

déclarations de constructeurs

déclarations de méthodes publiques

```
private: };
```

class et struct sont « identiques » autre l'accessibilité par défaut des membres

L'accessibilité est contrôlée par le langage

déclarations de méthodes privées

attributs

Exemple de définition de classe

```
class Employee
{
public:
    Employee();
    Employee(const string& name, double salary);
    double getSalary() const;           const après les parenthèses s'applique à this
    const string& getName() const;
    void setSalary(double salary);

private:
    string name_;
    double salary_;
};
```

Les constructeurs ont le nom de la classe.

this (équivalent du *self* Python) est implicite

const après les parenthèses s'applique à *this*

Remarque: par convention, nous utiliserons un *_* pour distinguer les variables qui correspondent aux attributs d'une classe.

Création d'un objet

Syntaxe similaire à Python

```
Employee marcel = Employee("Marcel", 50000);
```

Classe de l'objet

Construction de l'objet avec paramètres

```
Employee marcel("Marcel", 50000);
```

Syntaxe abrégée

```
Employee marcel{"Marcel", 50000};
```

Sans conversion restrictive implicite (C++11)

```
Employee marcel = Employee();
```

Construction par défaut

```
Employee marcel;
```

Construction par défaut, syntaxe abrégée; pas de ()

Implémentation des méthodes

- Méthodes généralement implémentées séparément de la définition de classe, en C++

```
typeRetour NomClasse::nomFonction(param1, ...)  
{  
    instructions  
}  
}
```

Vu plus tard

- Parfois dans la classe, surtout si *template*

```
class NomClasse {  
    ...  
    typeRetour nomFonction(param1, ...) {  
        instructions  
    }  
}
```

{ } avec le corps de méthode au lieu du ;

Exemple d'implémentation d'une méthode

```
void Employee::setSalary(double salary)
{
    if (salary > salary_)
        salary_ = salary;
}
```

Paramètre *this* implicite

on accède aux attributs de *this* implicitement

Paramètres d'une méthode

- Quand on écrit:
 - `marcel.setSalary(55000);`
- le compilateur fait appel à une fonction avec deux paramètres:
 - `nom_bizarre_de_fonction(marcel, 55000);`
- Mais tout cela est transparent pour nous

« name mangling » vu en INF1600

Interface d'une classe (rappel)

- L'interface d'une classe comprend toutes les fonctions publiques dont les items suivants:
 - Les constructeurs (toute classe devrait posséder au moins un constructeur par défaut, qui ne prend aucun paramètre)
 - Les fonctions de modification (mutators)
 - Les fonctions d'accès (accessors)
 - Les fonctions utiles à l'objet

Principe d'encapsulation (rappel)

- On n'a pas accès directement aux attributs d'un objet
- On modifie ou on obtient la valeur d'un attribut toujours par l'intermédiaire d'une méthode
- En résumé, on ne peut manipuler l'état d'un objet que par des méthodes qui sont définies par sa classe (interface)

Constructeur

- Le rôle principal d'un constructeur est d'initialiser les attributs lors de la création d'un objet
- En général, on a un constructeur par défaut, qui ne reçoit aucun paramètre, et qui donne aux attributs des valeurs par défaut
- On peut aussi avoir des constructeurs qui acceptent comme paramètres les valeurs initiales que l'on veut donner aux attributs

Exemple de constructeur par défaut et avec paramètres

```
Employee::Employee()  
{  
    name_ = "unknown";  
    salary_ = 0.0;  
}
```

```
Employee::Employee(const string& name, double salary)  
{  
    name_ = name;  
    salary_ = salary;  
}
```

Valeurs par défaut

- Les paramètres d'une fonction/méthode peuvent avoir des valeurs par défaut
 - Un constructeur où tous les paramètres ont des valeurs par défaut est un constructeur par défaut

```
class Employee
{
public:
    Employee(const string& name = "unknown",
              double salary = 0.0);
    ...
};
```

Peut être appelé avec 0, 1 ou 2 paramètres, comme en Python.

Quand un constructeur par défaut est-il appelé?

- Lors de la simple déclaration d'une variable d'objet:
 - `Employee p;`
- Lorsqu'on utilise un tableau d'objets:
 - `Employee tabDeEmployee[10];`

Constructeur par défaut appelé pour chaque objet du tableau

Quand un constructeur par défaut est-il appelé? (suite)

- Lorsque l'objet est lui-même un attribut d'un autre objet:

```
class Company
{
public:
    Company ();
    ...
private:
    Employee president_;
    ...
    int nbEmployees_;
};
```

```
Company::Company()
{
    nbEmployees_ = 0;
}
```

Pour initialiser cet attribut, le constructeur par défaut de *Employee* sera appelé, **avant** le constructeur de *Company*

Liste d'initialisation

- Comment contrôler la construction des membres si elle se fait avant le constructeur de cet objet?

```
class Company
{
public:
    Company ();
    ...
private:
    Employee president_;
    ...
    int nbEmployees_;
};
```

Liste d'initialisation spécifie la construction des membres

```
Company::Company() :
    president_("La presidente"),
    nbEmployees_(0)
{
    // rien à faire d'autre
}
```

Initialisation des attributs (C++11)

- On peut spécifier la construction par défaut directement sur les membres
 - Liste d'initialisation est prioritaire sur ces constructions

```
class Company
{
public:
    ...
private:
    Employee president_ = Employee("La presidente");
    // ou Employee president_{"La presidente"};
    ...
    int nbEmployees_ = 0;
};
```

Pas: Employee president("La presidente");
(serait vu comme une fonction)

Initialisation peut être une expression, ex.:
int* valeurs_ = new int[nValeurs_];

Délégation de constructeur (C++11)

```
Employee::Employee() : Employee("unknown", 0.0) {  
}
```

Utilise l'autre constructeur comme base avant le corps de ce constructeur (ici vide); les membres sont alors tous construits, **pas possible d'ajouter une liste d'initialisation**

```
Employee::Employee(const string& name, double salary) :  
    name_(name),  
    salary_(salary)  
{  
}
```

Destructeur

- Un destructeur est une méthode qui est appelée lorsqu'un objet est détruit
- On s'en sert pour faire du ménage
 - Par exemple, c'est dans le destructeur qui désallouera des pointeurs appartenant à l'objet en question
- A le même nom que la classe précédé d'un tilde ~

```
Employee::~Employee()  
{  
    // Rien à faire dans ce cas-ci  
}
```

Quand un destructeur est-il appelé?

- Quand la variable (ou constante) est détruite
 - Variable dans une fonction: à l'accolade } ou return
 - Variable globale: après la fin du *main()*
 - Destruction faite en ordre inverse de construction
- Si l'objet est créé dynamiquement (avec *new*), le destructeur est appelé lors du *delete*, juste avant de relâcher la mémoire allouée
 - Important que new avec [] ait delete avec [] (tous les destructeurs d'un tableau sont appelés)

Méthodes *const*

- Certaines méthodes ne servent qu'à obtenir l'état ou les attributs d'un objet
- On aimeraient que ces méthodes ne puissent effectuer aucune modification à l'objet
- Mot clé *const* après l'en-tête de la méthode
 - dans la définition de la classe
 - et dans l'implémentation (si séparée)

Exemple de fonction d'accès

```
const string& Employee::getName() const
{
    // name_ = "pierre";
    return name_;
}
```

Erreur de si la fonction tente de modifier la valeur d'un attribut de l'objet **ou si la fonction utilise une méthode qui n'a pas été déclarée constante.**

Exemple de code erroné avec *const*

```
class Employee
{
public:
    Employee(string name =
              "unknown", double salary
              = 0);
    double getSalary();
    string getName();
    ...
    void print() const;

private:
    string name_;
    double salary_;
};

string Employee::getName()
{
    return name_;
}

double Employee::getSalary()
{
    return salary_;
}

void Employee::print() const
{
    cout << getName() << endl;
    cout << getSalary() << endl;
}
```

Erreur!
Pourquoi?

Exemple de code erroné avec *const*

```
class Employee
{
public:
    Employee(string name =
              "unknown", double salary
              = 0);
    double getSalary() const;
    string getName() const;
    ...
    void print() const;

private:
    string name_;
    double salary_;
};
```

```
string Employee::getName() const
{
    return name_;
}

double Employee::getSalary() const
{
    return salary_;
}

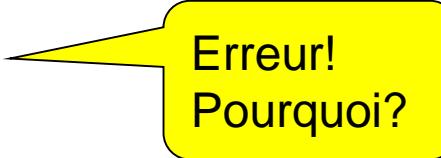
void Employee::print() const
{
    cout << getName() << endl;
    cout << getSalary() << endl;
}
```

Encore un exemple de code erroné avec *const*

```
class Employee
{
    ...
    void print();
    ...
};

void Company::print() const
{
    ...
    president_.print();
    ...
}
```

```
class Company
{
public:
    ...
    void print() const;
private:
    ...
    Employee president_;
};
```



Erreur!
Pourquoi?

Encore un exemple de code erroné avec *const*

```
class Employee  
{  
    ...  
    void print();  
    ...  
};
```

Cette méthode n'est pas déclarée constante.

```
class Company  
{  
public:  
    ...  
    void print() const;  
private:  
    ...  
    Employee president_;  
};
```

Cette méthode n'a pas le droit de modifier un attribut.

```
void Company::print() const  
{  
    ...  
    president_.print();  
    ...  
}
```

La méthode *print* n'est pas déclarée *const*, elle pourrait donc modifier l'attribut *president* (même si elle ne le fait pas en pratique).
Erreur de compilation.

Encore un exemple de code erroné avec *const*

```
class Employee
{
    ...
    void print() const;
    ...
};

void Company::print() const
{
    ...
    president_.print();
    ...
}
```

```
class Company
{
public:
    ...
    void print() const;
private:
    ...
    Employee president_;
};
```

Tuples

pair, liaison structurée, tuple

Motivation

- Comment retourner plusieurs valeurs d'une fonction?
 - *vector* ou *array* si valeurs de même type
 - *struct/class* : un nouveau type à chaque fois?
 - Paramètres out : souvent moins lisible
- Comment avoir l'équivalent d'`enumerate` Python?
 - `for i,x in enumerate(conteneur):` *Python*
 - `for (auto&& [i, x] : enumerate(conteneur))` *C++* cppitertools
 - *enumerate* donne des éléments contenant 2 valeurs de types possiblement différents

std::pair

⇒ Défini dans <utility>

- Classe utilitaire générique pour regrouper deux éléments hétérogènes
- Construction / affectation :
 - `pair<int, string> a(4, "bonjour");`
 - `pair b(5, "les amis");` C++17, types selon arguments:
*int, const char**
 - `a = pair<int, string>(5, "salut");`
 - `a = pair(5, "hello");` C++17
 - `a = make_pair(6, "bye");` Utile avant C++17
 - `a = b;` Types différents mais conversions implicites existent
 - `// b = a;` Non: n'existe pas conversion implicite de *string* vers *const char**

Liaison structurée (« structured binding »)

```
pair<int, double> f(int i) {
    return { i*i, sqrt(i) }; pair implicite car c'est le type de retour
}

int main() {
    pair p = f(2);
    cout << p.first << " " << p.second << endl; //? first et second
        accessibles mais ne sont pas des noms significatifs

    auto [racine, auCarre] = f(2); //? Liaison structurée donne des noms
    cout << racine << " " << auCarre; Toujours auto... on ne peut pas indiquer deux types

    array t = { 2.3, 4.5, 5.6 };
    for (auto ix : enumerate(t)) //? ix d'une classe qui hérite de pair
        cout << ix.first << ":" << ix.second << endl;

    for (auto [i, x] : enumerate(t)) //? Liaison structurée
        cout << i << ":" << x << endl;
}
```

std::tuple

C++11

⇒ Défini dans <tuple>

- *pair* pas pratique pour autre que deux :
 - pair<int, pair<double, string>> p;
 - p.second.second = "allo";
 - Liaison structurée sur seulement un niveau à la fois
- *tuple* regroupe plusieurs éléments hétérogènes :
 - tuple<int, double> a;
 - tuple<int, double, string> b;
 - tuple<int, double, string, char> c;
 - // ...
 - auto [i, valeur, nom] = b; Liaison structurée fonctionne

std::tuple

- Construction / affectation similaire à *pair* :

- `tuple<int, double, string> a(4, 1.2, "bonjour");`

- `tuple b(5, 2.3, "les amis");` C++17

- `a = tuple<int, double, string>(5, 4.5, "salut");`

- `a = tuple(5, 6.7, "hello");` C++17

- `a = make_tuple(6, 7.8, "bye");` Utile avant C++17

- `a = b;` Types différents mais conversions implicites existent

- Accès avec fonction globale *get* indice constant ou type (si unique) :

- `cout << get<2>(a);` *<constante à la compilation>*

Pas [] comme Python

- `get<1>(a) = 2.4;` Utilisable en écriture

- `cout << get<string>(a);` Possible si un seul *string* dans le *tuple*

- `get<double>(a) = 2.4;`

pair et tuple

- Comparaisons entre mêmes types

- == !=
 - Vérifie composante à composante, utilisant ==
- < <= > >=
 - Selon ordre lexicographique, utilisant < ou <=
- Tous les types contenus doivent supporter ces opérateurs pour que l'opérateur fonctionne

- Utilisable comme clé de *map*

- Mais pas directement dans *unordered_map*

Similaires à *dict* Python

Pointeurs intelligents

unique_ptr, shared_ptr

Définition

- Un pointeur brut = pointeur dont le type est directement T^* (référence brut = $T\&$)
- Un pointeur intelligent est une classe qui encapsule la notion de pointeur et règle l'appartenance de la mémoire dynamique.

Motivation

- Pointeur brut (T^*) : source majeure de problèmes.
 - Qui doit désallouer la mémoire dynamique?
 - Lors d'un passage en paramètre
 - Lorsque retourné par une fonction
 - Dans un simple new (i.e. items dans Qt)
 - Oublie de désallouer → fuite de mémoire.
 - Utilisation après la désallocation → « undefined behavior ».
 - Comment bien gérer les exceptions?
 - Quand désallouer la mémoire d'une possession de mémoire partagée?

Bonnes pratiques en C++

- Pour ces raisons: (C++ core guidelines)
 - Jamais transférer la possession de la mémoire à l'aide d'un pointeur ou référence brut (I.11)
 - Un pointeur ou référence brut ne devrait jamais posséder une ressource mémoire (R.3, R.4)

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

Pointeurs intelligents (C++11)

- But:
 - Désallocation automatique de la mémoire quand aucun pointeur intelligent n'y pointe.
 - Incluant si son propriétaire détruit par une exception
 - Utilisation comme un pointeur
- Propriétaire unique: `unique_ptr`
 - L'unique pointeur intelligent décide de la durée de vie de l'espace mémoire
- Plusieurs propriétaires: `shared_ptr`
 - Plusieurs pointeurs intelligents possèdent la mémoire dynamique
- `#include <memory>`

unique_ptr et shared_ptr

- (« ... » pour « unique » ou « shared »)
- Classes génériques
 - `..._ptr<T>` un élément de type T
 - `..._ptr<T[]>` tableau d'éléments
- Surcharge pour utilisation comme un pointeur
 - `* ->` s'il pointe vers un élément
 - `[]` s'il pointe vers un tableau
 - `=` dont `= nullptr`
 - `== != < <= > >=` comparaison de pointeurs
- Généralement préférer `make_...` à `new` pour allouer la mémoire dynamiquement
 - `make_...<T>(arguments, au, constructeur)`
 - `make_...<T[]>(taille_du_tableau)`
 - (`make_unique` est C++14; `make_shared<T[]>` est C++20)

unique_ptr

- Ne peut pas être copié
 - Erreur de compilation lors de copie
 - Un passage par valeur sans « move » est une copie donc une erreur de compilation
- Peut transférer la possession de la mémoire dynamique
 - Utiliser std::move

```
vector<unique_ptr<Item>> items;
auto unItem = make_unique<Item>();
items.push_back(move(unItem));
```

Pourquoi?

Allocation dynamique.

Transfert possession au vector.
unItem devient nullptr.

unique_ptr – Exemple

● Avant

```
class MaClasse {  
public:  void posseder(Item* item) {  
    delete item_; ——————  
    item_ = item;  
}  
~MaClasse() { delete item_; }  
private: Item* item_ = nullptr;  
};
```

Pas évident que l'appelant donne sa possession de la mémoire.

Doit libérer manuellement, sinon fuite

Doit initialiser sinon « undefined behavior » lors du premier delete.

● Avec unique_ptr

```
class MaClasse {  
public:  void posseder(unique_ptr<Item> item)  
    { item_ = move(item); }  
private: unique_ptr<Item> item_;  
};
```

Appelant doit donner sa possession

Désallocation automatique de l'ancien item_, et item_ conserve l'espace mémoire du paramètre item

Désallocation automatique à la destruction

std::unique_ptr – Exemple (suite)

- Avant

```
MaClasse a;  
Item* item = new Item(1, "sans1");  
cout << item->nom << endl;  
a.posseder(item); Manque-t-il un delete après?  
a.posseder(new Item(2, "sans2"));
```

- Avec unique_ptr

```
MaClasse a;  
auto item = make_unique<Item>(1, "avec1");  
cout << item->nom << endl; Comme un pointeur  
a.posseder(move(item)); Transfert explicite; item est nullptr après  
a.posseder(make_unique<Item>(2, "avec2"))); move non nécessaire  
car objet temporaire
```

shared_ptr

En Python, toute variable est similaire à shared_ptr

- Possession de la mémoire est partagée entre différents pointeurs intelligents
 - Compteur de pointeurs intelligents qui possèdent la mémoire dynamique
 - `.use_count()` pour obtenir ce compte
 - Chaque copie compte +1, chaque destruction -1
 - **Mémoire dynamique libérée lorsque le dernier pointeur intelligent est détruit** (compte à 0)
- Peut transférer la possession (optimisation)
 - À un autre shared_ptr (**pas à un unique_ptr**)

```
vector<shared_ptr<Item>> items;
auto unItem = make_shared<Item>();
items.push_back(unItem);
items.push_back(move(unItem));
```

Partage la possession (+1)

Transfert la possession →
rien à compter;
unItem devient nullptr

shared_ptr

- Code :

```
using namespace std;  
  
int main()  
{  
    shared_ptr<int> ptr = make_shared<int>();  
    cout << "Nb : " << ptr.use_count() << endl;  
    shared_ptr<int> ptr2 = ptr;  
    cout << "Nb : " << ptr.use_count() << endl;  
}
```

- Affichage :

Nb : 1

Nb : 2

Bonnes pratiques en C++ (suite)

- On devrait: (C++ core guidelines)
 - Utiliser `unique_ptr` et `shared_ptr` pour représenter la possession mémoire. (R.20)
 - Préférer `unique_ptr` à `shared_ptr` sauf si besoin de partager la possession. (R.21)
 - Prendre en paramètre à une fonction un pointeur intelligent si et seulement si pour changer la durée de vie de l'espace mémoire. (R.30)
 - Utiliser une simple référence (`T&`) ou pointeur (`T*`) si la fonction n'a pas à influencer la durée de vie.
 - Utiliser `T*` si peut être `nullptr`; sinon préférer `T&`.

Exemple shared_ptr

```
class Chanson {  
public:  
    Chanson(string chanteur): chanteur_(chanteur)  
    { cout << chanteur_ << endl; }  
    ~Chanson () { cout << "detruit "  
                  << chanteur_ << endl; }  
    string getChanteur() { return chanteur_; }  
private:  
    string chanteur_;  
};
```

Exemple shared_ptr

```
int main()
{  vector<shared_ptr<Chanson>> v {
    make_shared<Chanson>("Bob Dylan"),
    make_shared<Chanson>("Aretha Franklin"),
    make_shared<Chanson>("Celine Dion")
};

vector<shared_ptr<Chanson>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2),
               [](const shared_ptr<Chanson>& s)
               { return s->getChanteur() == "Bob Dylan"; });
```

Les chansons sont partagées entre les v et v2

Exemple shared_ptr

```
cout << "V" << endl;  
for (const auto& s : v)  
    cout << s->getChanteur() << endl;
```

```
cout << "V2" << endl;  
for (const auto& s : v2)  
    cout << s->getChanteur() << endl;
```

Bonnes pratiques – Exemple

```
void parametreParReference(Item& item)
{
    item.nom += "A";
}

void parametreParPointeur(Item* item)
{
    if (item != nullptr)
        item->nom += "B";
}

int main()
{
    vector<unique_ptr<Item>> items;
    //...
    parametreParReference(*items[1]);
    parametreParPointeur(items[2].get());
}
```

unique_ptr ou shared_ptr ne change rien au reste du code de cet exemple

.get() pour obtenir le pointeur

Pointeurs intelligents et itération sur conteneurs STL

- Doit utiliser une référence au pointeur

```
vector<unique_ptr<Item>> items;
```

Référence au pointeur intelligent;
pas de copie

```
for (const auto& item : items)  
    cout << item->nom << ";" ;
```

Se comporte comme un pointeur
(-> et non .)

std::shared_ptr – Problème de références circulaires

- Si A réfère à B et B réfère à A (par agrégation)

```
struct S {
    shared_ptr<S> autre; Pointeur vers un autre de même type
    ~S() { cout << "~S\n"; } Pour voir s'ils sont détruits
};
shared_ptr<S> variable;
{
    auto a = make_shared<S>(), b = make_shared<S>(); Alloue A et B
    a->autre = b; b->autre = a; Chacun réfère à l'autre
    variable = a; Conserve un pointeur vers A
}
cout << "On enlève le pointeur...\n";
variable = nullptr; A et B ne sont plus accessibles.  
Sont-ils détruits?
```

- Le compte est 1 pour A et B, donc non.
 - (on évitera ces structures dans ce cours)

En Python, le « garbage collector » vérifie régulièrement, ou sur demande: gc.collect()

Résumé

- Un pointeur intelligent est une classe générique.
- La mémoire dynamique allouée est détruite quand le pointeur intelligent qui possède cette mémoire n'y pointe plus (détruit ou réaffecté).
- Cette mémoire dynamique peut ou ne pas être partagée.
- move() pour le transfert de possession de mémoire à un autre pointeur intelligent.

Surcharge

d'opérateurs, pour flux d'entrée/sortie,
appels en cascade, de méthode,
« tag dispatching »

Opérations arithmétiques sur des objets

- Comment permettre:

- Fraction a, b; cin >> a >> b;
- Fraction c = a + b;
- cout << c;

Fonctionne avec *int* ou *double*, pourquoi pas Fraction, Complexe, Matrice ... ?

- Nouvelles fonctionnalités aux opérateurs

- $1 \ll 4$ est 1 décalé à gauche de 4 bits = 16
- cout << "bonjour" affiche

En C, C++, et Python

- Attention: nuit la lisibilité si contre intuitif

- Ex: $a + b$ qui modifie a

La classe Fraction

```
class Fraction
{
public:
    Fraction(int numerateur = 0, int denominateur = 1);
    int getNumerateur() const;
    int getDenominateur() const;
    double getReel() const;

private:
    int numerateur_;
    int denominateur_;
    void simplifier();
};
```

Surcharges d'opérateurs

- **Redéfinir la fonctionnalité d'un opérateur pour une classe.**
 - + - * / % < == += etc.
 - Ne peut pas créer de nouveaux opérateurs
- **L'ordre de priorité est conservé.**
 - && || perdent l'évaluation court-circuit
- **Les opérateurs**
 - Nombre déterminé de paramètres (binaires et unaires)
 - Sauf () qui permet un nombre quelconque
 - Ne peuvent pas avoir de paramètres par défaut
 - < <= > >= peuvent utiliser <=>, != peut utiliser ==
 - Peuvent renverser les paramètres, ex: a == b peut b.operator==(a)

a += b indépendant de a + b, pas comme Python

C++20

Surcharges d'opérateurs

Peuvent être surchargés

`++ -- () [] ->`
`+ - ! ~ * & (unaires)`
`+ - * / %`
`<< >>`
`< <= > >= == != <=>`
`& ^ | && ||`
`= += -= *= /= %= <<= >>=`
`&= ^= |=`
`,`
`new delete`

C++20

Ne peuvent pas

`. :: ?: .* sizeof typeid`

Non-membre: `= () [] ->`

Doivent être membre

Opérateur binaire

- Opérandes (arguments d'un opérateur) à gauche et droite de l'opérateur
 - Ex: $a + b$
 - cherche $a.operator+(b)$ (*this* à gauche du +)
 - sinon $operator+(a, b)$
- Exemple Fraction:
 - Fraction f1(1,2), f2(4,5);
 - Fraction f3 = f1 + f2; // f1.**operator+**(f2);
 - Où operator+() est membre de classe Fraction
 - car f1 est une Fraction
 - Retourne un objet Fraction

La classe avec Fraction + Fraction

```
class Fraction
{
public:
    Fraction(int numerateur = 0, int denominateur = 1);
    int getNumerateur() const;
    int getDenominateur() const;
    double getReel() const;
    Fraction operator+ (const Fraction& autre) const;

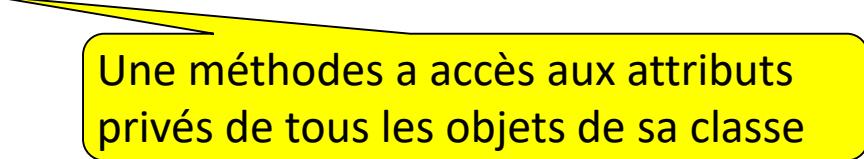
private:
    int numerateur_;
    int denominateur_;
    void simplifier();
};
```

Notre + ne modifie pas les opérandes

Fraction + Fraction (méthode membre)

```
Fraction Fraction::operator+(const Fraction& autre) const
{
    return {
        numerateur_ * autre.denominateur_ +
        autre.numerateur_ * denominateur_,
        denominateur_ * autre.denominateur_
    };
}
```

 La construction simplifie la Fraction.

 Une méthode a accès aux attributs privés de tous les objets de sa classe

Fraction + int

- Fraction + Fraction fonctionne
 - Fraction f1(1,2), f2(1,4);
 - Fraction f3 = f1 + f2; // Fait f1.operator+ (f2)
- Fraction + int fonctionne aussi
 - Fraction f4 = f1 + 1; // Fait f1.operator+ (Fraction(1))
 - Trouve l'opérateur qui prend une Fraction
 - Tente de convertir int en Fraction: trouve un constructeur
 - Attention: f1 + 1.2 // Fait f1.operator+ (Fraction(1))
 - Peut l'empêcher
- Peut optimiser une méthode pour Fraction + int

Fraction + int optimisée

Doit ajouter aussi la déclaration dans la classe

```
Fraction Fraction::operator+ (int entier) const
{
    return {
        numerateur_ + entier * denominateur_,
        denominateur_
    };
}
```

Deux multiplications de moins

Mais la construction simplifie la Fraction inutilement.
Pourrait avoir un constructeur spécial qui ne simplifie pas.

// Autre manière :

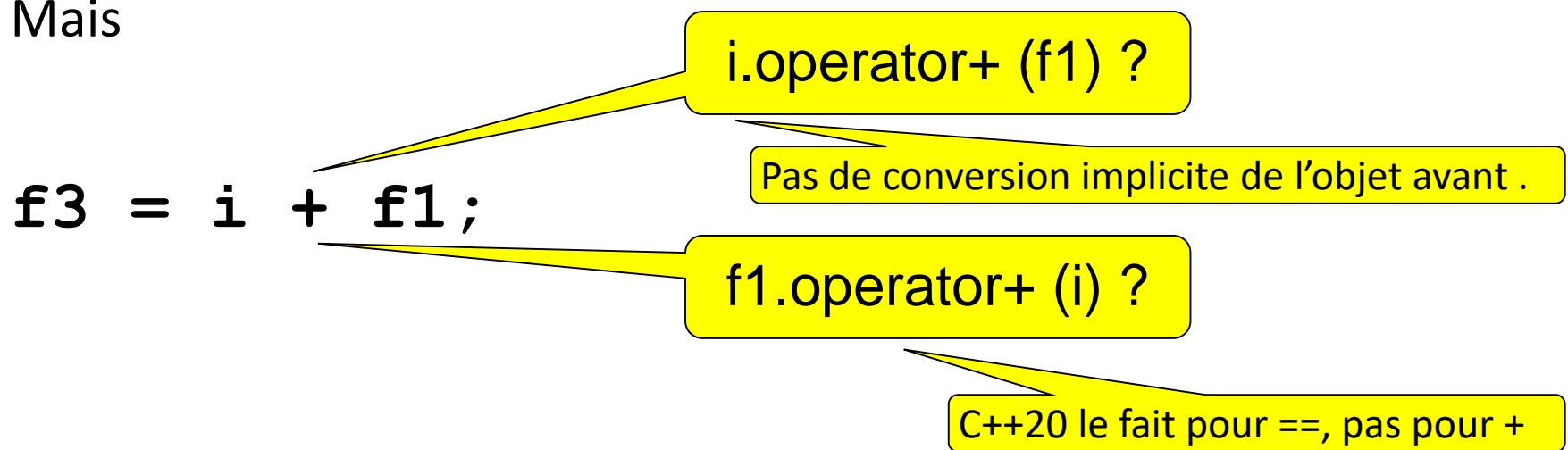
```
Fraction Fraction::operator+ (int entier) const
{
    Fraction somme = *this; Accès explicite à this
    somme.numerateur_ += entier * denominateur_;
    return somme;
}
```

Copie ne simplifie pas la Fraction

Opérateur + **n'est PAS** commutatif!

```
Fraction f1(1,2), f2(1,4);  
int i = 1;  
Fraction f3 = f1 + i; // f1.operator+ (i);  
// Utilise Fraction::operator+ (int) const
```

Mais



Solution: fonctions globales

```
class Fraction{  
public:  
    ...  
    Fraction operator+(const Fraction&) const;  
    Fraction operator+(int) const;  
private:  
    ...  
};  
Fraction operator+ (int, const Fraction&);
```

.h

pas une fonction membre

```
Fraction operator+ (int e, const Fraction& f){  
    return f + e;  
}
```

.cpp

Si besoin d'accès membres privés: friend

```
class Fraction{  
public:  
    ...  
    Fraction operator+(const Fraction&) const;  
    Fraction operator+(int) const;
```

.h

```
    friend Fraction operator+(int, const Fraction&);  
private:  
    ...  
};
```

fonction globale avec accès
aux membres privés

Pourquoi pas const ?

```
Fraction operator+ (int e, const Fraction& f){  
    // peut accéder f.numerateur_ f.denominateur_ ...  
}
```

.cpp

friend détruit-il l'encapsulation ? **NON**

- Des fonctions non-membres ont accès aux attributs privés, mais...
- La classe déclare ses fonctions *friend*
 - Donc similaire à une méthode sans *this* implicite
- Les opérateurs font partie de l'interface d'une classe, que l'objet soit à gauche ou droite
- Mieux qu'ajouter des accesseurs publiques juste pour l'opérateur
- **Ne pas** ajouter *friend* pour contourner un problème de conception des classes

Surcharge de flux d'entrée et de sortie

- On veut
 - Fraction f1, f2; cin >> f1 >> f2;
 - cout << f1 << ", " << f2 << endl;
- On n'ajoute pas des méthodes à cin/cout
 - Bibliothèque pas écrite par nous
 - La présentation d'un objet est plus reliée à sa classe qu'à celle du flux
- L'opérateur doit
 - être global (ou *friend*)
 - retourner une référence au stream pour permettre les cascades: cout << f1 << endl;

cin/cout sont à gauche de l'opérateur

Associativité: (cout << f1) << endl;

Surcharge de flux de sortie

```
class Fraction {  
public:  
    ...  
    friend ostream& operator<< (ostream& o, const Fraction& f);  
    ...  
};
```

.h

```
private:
```

```
    ...
```

Référence au ostream (cout, ofstream, ...)

Référence au même ostream

```
ostream& operator<< (ostream& o, const Fraction& f) {  
    return o << f.numerateur_ << "/" << f.denominateur_;  
}
```

.cpp

Accès aux attributs privés

Cascade se généralise (pas juste opérateurs << et >>)

```
class Fraction {  
public:  
    ...  
    Fraction& incrementer();  
    ...  
};  
  
Fraction& Fraction::incrementer() {  
    numerateur_ += denominateur_;  
    return *this;  
}
```

Permet: f1.incrementer().incrementer().incrementer();

Surcharge d'opérateur unaire

```
class Fraction {  
public:  
    ...  
    Fraction& operator++ (); // ++f; (prefix)  
    Fraction operator++ (int); // f++; (postfix)  
    ...  
};
```

Pas de paramètre

Dummy (toujours *int*, pas de nom)

.h

```
Fraction& Fraction::operator++ () {  
    return incrementer();
```

Retourne l'objet mis à jour

.cpp

```
    }  
    Fraction Fraction::operator++ (int) {  
        Fraction original = *this;  
        ++(*this);
```

Met à jour l'objet

```
        return original;
```

Retourne l'original avant l'incrément

}

Surcharge : même « nom » types différents (pas juste pour opérateurs)

« name mangling » en INF1600

```
void f(int a);
```

nombre différent
de paramètres

```
void f(int a, double b);
```

types différents
paramètres

```
void f(int a, double b, int c);
```

```
void f(int a, double b, string s);
```

Attention aux
conversions
implicites

```
void f(int a, double b, Fraction f);
```

```
// int f(int a);
```

Ne peut pas être distinguée
uniquement par le type de retour

« tag dispatching » ou aiguillage par étiquette

- Un type vide est utilisé pour distinguer une surcharge ayant autrement tous les même types
 - Souvent pour des constructeurs, car on ne peut pas les distinguer en utilisant un autre nom de fonction

```
Fraction Fraction::operator+ (int entier) const
{
    return { SansSimplifier, Objet d'un type vide
        numerateur_ + entier * denominateur_,
        denominateur_
    };
}
```

Exemple de construction de Fraction avec aiguillage par étiquette

```
class Fraction
{
    ...
    struct SansSimplifier_t{};
    static constexpr SansSimplifier_t SansSimplifier{};

    Fraction(SansSimplifier_t, int numerateur, int denominateur);
    ...
};

...
Fraction::Fraction(SansSimplifier_t,
                    int numerateur, int denominateur) :
    numerateur_(numerateur),
    denominateur_(denominateur)
{}
```

Type vide

Objet vide constant à la compilation

Type sans nom de paramètre, juste pour le choix de surcharge

Copie

Composition vs agrégation,
constructeur de copie, opérateur =,
« move »

Copie, référence/pointeur : Python vs C++ (rappel)

Python

- Toutes les variables sont des références
- L'affectation copie la référence

```
import copy
a = [1]
x = a; y = copy.copy(a);
x[0] = 2; # a contient {2}
y[0] = 3; # a non modifié
```

Sans bibliothèque

Référence,
pas copie

- copy ou deepcopy pour copier*

C++

- Les variables sont directement du type (statique) indiqué
- L'affectation (pas la création de référence) copie la variable

```
struct Arr1 { int valeurs[1]; };
Arr1 a = {1}; auto b = &a;
auto& x = a; auto y = a; auto z = b;
x.valeurs[0] = 2; // a contient {2}
y.valeurs[0] = 3; // a non modifié
z->valeurs[0] = 4; // a contient {4}
```

Copie le pointeur

Comment implémenter vector ?

- La taille d'un type C/C++ est toujours constante
 - `sizeof(x)` constante à la compilation = nombre d'octets
- → les données du *vector* ne sont pas dans *vector*
 - Il contient un pointeur vers les données
 - Taille d'allocation dynamique est spécifiée à l'exécution
- Première version la plus simple:
 - `struct Vecteur1 { int* valeurs; int nValeurs; };`
- Mais
 - **Vecteur1 a = {new int[1]{1}, 1}; vector b = {1};**
 - `auto x = a; x.valeurs[0] = 2;` // a modifié {2}
 - `auto y = b; y[0] = 2;` // b non modifié, comme pour Arr1
- *Vecteur1* et *vector* ne sont pas copiés de la même manière!
 - Pourtant, les deux contiennent un pointeur

L'exemple fait quoi
si *unique_ptr* ?

Constructeur de copie et affectation

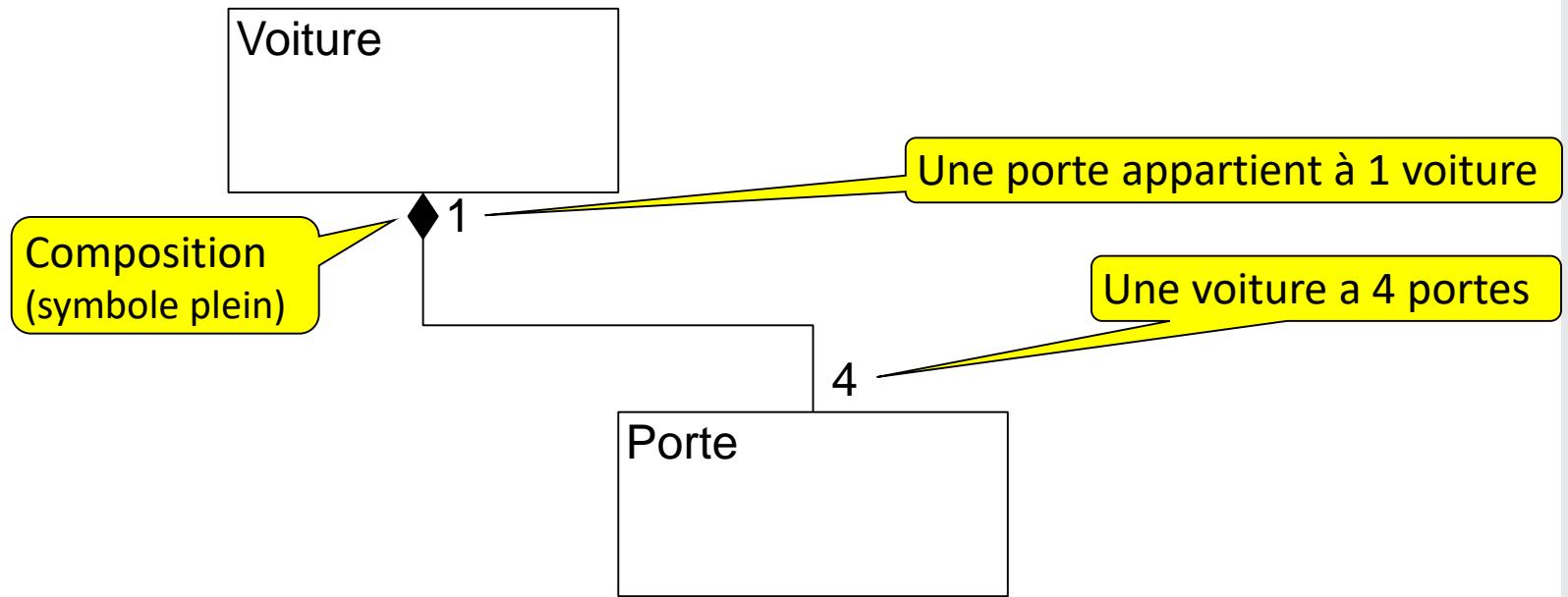
- On aimerait dire que les données sont dans le vecteur et donc qu'une copie du vecteur copie les données
 - Le pointeur est un détail technique d'implémentation
- Il faut définir **comment copier l'objet**
 - Copier les valeurs pointées
- Le constructeur de copie est utilisé quand un nouvel objet est créé en copiant un objet
 - Nouvelle variable, passage de paramètre par valeur, ou *new*
- L'opérateur= (affectation) est utilisé quand on affecte à un objet existant
- Par défaut ils copient juste les attributs eux-mêmes (« shallow copy ») pas les données référencées

Vecteur est
composé de
valeurs

Composition et agrégation (rappel et +)

- Distinction importante entre les deux en C++ pour
 - Copie: qu'est-ce qui est copié?
 - Possession/appartenance: qui détruit (désalloue) ?
- Composition: A est composé de B
 - Lien fort, un n'existe pas sans l'autre, B appartient à A
 - → copier A copie B, détruire A détruit B
- Agrégation: A est un agrégat de B
 - Lien plus faible, un regroupement, le/les individus de B continuent d'exister sans A
 - → copier A ne copie pas B, détruire A ne détruit pas B

Composition, représentation en UML

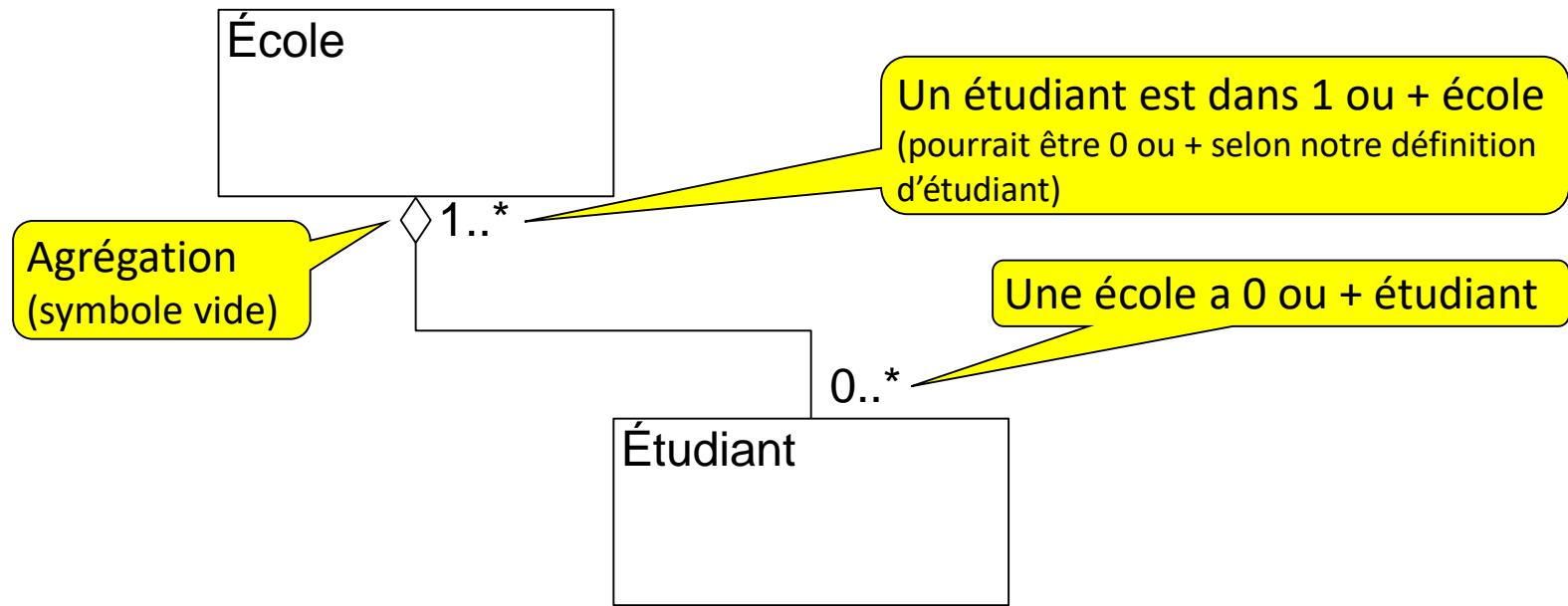


- Copier une voiture inclus la copie des portes
- Détruire une voiture détruit le portes
 - Peut être possible d'enlever les portes avant de détruire

Composition, implémentation

- A est composé de B
 - Directe: class A { ... B b; ... };
 - L'instance de B est « physiquement » dans A
 - $\text{sizeof}(A) = \dots + \text{sizeof}(b) + \dots$
 - Par pointeur: class A { ... B* b; ... };
 - Doit programmer la copie et la destruction
 - Par unique_ptr: class A { ... unique_ptr b; ... };
 - Doit programmer la copie
 - Pas par référence
 - Langage permet `delete &b;` mais est considéré de la mauvaise programmation.
- A ... b ...
- Taille en octets Pour B b[N]; $\text{sizeof}(b) = N * \text{sizeof}(B)$ Octets contigus en mémoire
- Ex: liste de listes Préférer `gsl::owner<B*>`
- Quelle est la copie par défaut?

Agrégation, représentation en UML



- Copier une école n'inclus pas les étudiants
- Détruire une école laisse les étudiants par eux-mêmes

Agrégation, implémentation

- A est un agrégat de B
 - ne pas copier ni détruire le B avec A
- Par référence: class A { ... B& b; ... };
- Par pointeur: class A { ... B* b; ... }; 
Pas gsl::owner
- Par shared_ptr: class A { ... shared_ptr b; ... };
- Copie et destruction par défaut fonctionnent
 - Destruction pourrait avoir à avertir *b* qu'il n'est plus dans cet agrégat

Conteneurs, composition et agrégation

- Un conteneur (array, vector, list ...) **contient** des cases/éléments
 - Est **composition** de cases/éléments
- Attention à ce qu'il contient
 - vector est composé d'objets B
 - vector<B*> est composé de pointeurs
 - Agrégation d'objets B, ou
 - Composition d'objets B si on ajoute copie et destruction des B
- Une vue (span, string_view, ranges::views) **réfère** à des cases/éléments appartenant à un autre
 - Est **agrégation** de cases/éléments

Constructeur de copie

- Déclaration: Classe(const Classe& autre);
- Par défaut « shallow »

- Bon pour composition directe ou agrégation
- Semblable à

```
struct Vecteur2 { int* valeurs; int nValeurs; }
```

Si on ne déclare pas ce constructeur

```
Vecteur2(const Vecteur2& autre) :  
    valeurs(autre.valeurs), nValeurs(autre.nValeurs) {}
```

Constructeur de copie « shallow »
copie les attributs

```
Vecteur2(int* v, int nV) :  
    valeurs(v), nValeurs(nV) {}  
};
```

Utilise le constructeur de copie des attributs

Déclarer un constructeur enlève la construction standard d'une struct avec {}, on ajoute donc un constructeur.

Constructeur de copie, pour composition non directe

- On veut une copie plus profonde (« deep »)

```
struct Vecteur3 { int* valeurs; int nValeurs;
    Vecteur3(const Vecteur3& autre) {
        valeurs = new int[nValeurs = autre.nValeurs];
    }
```

```
        for (int i = 0; i < nValeurs; i++)
            valeurs[i] = autre.valeurs[i];
```

On utilise ici l'operator= pour chaque valeur

constructeur de copie « deep »
alloue et copie les valeurs

...
};
...
On a fait un *new*, ça prendrait un destructeur qui *delete*
mais l'exemple ci-dessous va planter; pourquoi?

```
Vecteur3 a = {new int[1]{1}, 1};
auto x = a; x.valeurs[0] = 2; // a non modifié
x = a; x.valeurs[0] = 3;    // a contient {3}
```

Bien!

Oups!

Opérateur= affectation de copie, pour composition non directe

- On veut que = copie « deep » aussi

```
struct Vecteur4 { int* valeurs; int nValeurs;
    Vecteur4& operator=(const Vecteur4& autre) {
        if (this != &autre) {
            delete[] valeurs;
            valeurs = new int[nValeurs = autre.nValeurs];
            for (int i = 0; i < nValeurs; i++)
                valeurs[i] = autre.valeurs[i];
        }
        return *this;
    }
    Vecteur4(const Vecteur4& autre) : valeurs(nullptr), nValeurs(0) {
        *this = autre;
    }
    ~Vecteur4() { delete[] valeurs; }
    ...
}
```

Toujours vérifier qu'on ne copie pas dans lui-même: $v = v$;

Optimisation: on pourrait réallouer seulement si la taille n'est pas déjà bonne

Permet la cascade: $a = b = c$;

Nécessaire?

Maintenant $x = a$; copie bien a .

« move »

- Si une fonction retourne un Vecteur4 par valeur
 - Vecteur4 $f()$ { ... return a ; }
 - int main() { Vecteur4 x = ...; $x = f();$... }
 - L'affectation désalloue x , alloue et copie a dans x
 - Les variables de f sont détruites, donc a , qui désalloue a
- Une optimisation serait
 - Désalloue x , puis x prend possession du tableau de a
 - La destruction de a ne désalloue rien
- Un opérateur= de « move » le permet

Le pointeur
alloué dans ...

une allocation, copie et
désallocation de moins

C++11

Constructeur et opérateur= de « move »

```
struct Vecteur5 {
    int* valeurs; int nValeurs;
    ... copie comme Vecteur 4
```

&& référence à objet temporaire: sans nom, variable locale dans *return* ou en utilisant *move()*

Ne cause jamais d'exception

```
Vecteur5& operator= (Vecteur5&& autre) noexcept {
```

```
    delete[] valeurs;
```

Pas de vérification si on move dans lui-même

```
    valeurs = autre.valeurs; nValeurs = autre.nValeurs;
```

```
    autre.valeurs = nullptr;
```

Comme copie « shallow » ...

```
    return *this;
```

mais efface le pointeur de l'autre

```
}
```

```
Vecteur5(Vecteur5&& autre) noexcept : valeurs(nullptr) {
```

```
    *this = move(autre);
```

Réutilise l'opérateur de *move*

```
}
```

```
~Vecteur5() { delete[] valeurs; }
```

```
};
```

delete nullptr ne fait rien, lors de la destruction de *autre*

Copie interdite

- Certaines classes ne veulent pas de copie
 - Ex: unique_ptr
- Il suffit de déclarer construction/opérateur= « move » sans déclarer les « copie »
 - Déclarer un constructeur ou opérateur= de move enlève les versions par défaut du constructeur de copie et de move, et de l'opérateur= de copie et de move.
 - Message: « tentative de référencement d'une fonction supprimée » sur la fonction enlevée (constructeur ou opérateur=)

Bonne conception de Vecteur

- **Problèmes:**

- Le constructeur permet un état invalide
 - Si passe un tableau local au lieu d'allouer avec *new*
 - Si nValeurs ne correspond pas au tableau
- Prend possession d'un pointeur brut en paramètre
- L'accès direct aux membres permet un état invalide
- Gestion de *new/delete* compliquée

- **Solutions:**

- Constructeurs uniquement pour états valides
- Ne prendra pas possession d'un pointeur brut
- Membres privés et accesseurs : `operator[]` et `size()`
- Utiliser `unique_ptr` (« move » devient trivial)

Conception de std::vector (vs Vecteur5)

- std::vector permet l'ajout/retrait d'éléments à la fin
 - Capacité (taille allouée) \geq nombre d'éléments dans le vector
 - Réallocation seulement si capacité insuffisante
- Les cases allouées non utilisées n'ont pas fait de construction d'objet
 - Donc pas: `new TypeObjet[capacité]`
 - ni: `make_unique<TypeObjet[]>(capacité)`
 - (Pour type sans constructeur/destructeur, on peut faire ci-dessus)
- → il faut
 - Allouer/désallouer sans construire/détruire
 - `p = operator new(nOctets)` et `operator delete(p)`
 - ou `p = new byte[nOctets]` et `delete[] p`
 - Construire sans allouer: `new(pointeur) Type(arguments)`
 - Détruire sans désallouer: `pointeur->~Type()`

byte C++17, char
ou `unsigned char`

Dit placement new

Résumé

- Lors de la définition d'une classe C++, il faut toujours penser au minimum à
 - Est-ce qu'elle possède l'élément? (Composition ou agrégation)
 - Construction
 - par défaut
 - (avec paramètres)
 - Copie
 - constructeur de copie
 - opérateur=
 - Destruction
 - Et pour certaines optimisations, à « move »
 - constructeur
 - opérateur=
- Après y avoir pensé, on peut trouver que ce qui est fait par défaut est ce qu'on veut.
- `X()`
- Ou avec paramètres ayant tous des valeurs par défaut
- `X(const X& autre)`
`X& operator= (const X& autre)`
`~X()`
- `X(X&& autre) noexcept`
`X& operator= (X&& autre) noexcept`

Classes et fonctions génériques

Modèle (template) de fonction,
de classe, de méthode,
contrainte de type, traits de type

Motivation

- On a dans un programme:
 - class ListeEntiers { int* valeurs; int nValeurs; ... };
 - class ListeTextes { string* valeurs; int nValeurs; ... };
 - (avec mêmes méthodes)
 - Autre que le type des valeurs, c'est une duplication de code
 - pas d'héritage entre int/string donc pas *polymorphisme par sous-typage*
 - Les modèles (*template*) de classes et fonctions les rendent plus génériques en paramétrant
 - les types
 - les constantes à la compilation
- *template <typename T> class ListeValeurs { T* valeurs; int nValeurs; ... };*
- Modèle compilé pour chaque « valeur » utilisée de paramètres
 - dit *polymorphisme paramétrique* ou *polymorphisme de compilation*

Exemple de fonction générique

template < paramètres, du, template >

```
template <typename T>
void afficher(ostream& os, T valeurs[], int nValeurs)
{
    os << "{ ";
    for (int i = 0; i < nValeurs; i++) {
        if (i != 0)
            os << ", ";
        os << valeurs[i];
    }
    os << " }";
}
```

Le paramètre T est un type

Éléments du tableau de type T

Fonctionne seulement si << défini pour T;
sinon erreur de compilation ici générée par
l'appel d'afficher

<< fonctionne pour plusieurs type par *surcharge*
(polymorphisme ad hoc)

Utilisation:

```
int a[] = { 4, 1, 3 }; int b[] = { 5, 6 }; string c[] = { "b", "a" };
afficher(cout, a, 3); afficher(cout, b, 2); afficher(cout, c, 2);
```

Fonction compilée
2 fois (pour *int* et
string)

Le compilateur trouve le type « automatiquement », si possible

```
template <typename T>
void f() { /* ... */ }
```

```
template <typename T>
void g(const T& x) { /* ... */ }
```

```
template <typename T>
void h(span<T> x) { /* ... */ }
```

```
int main() {
    //f();           //NON Quel est T?
    f<int>();      // Choix explicite de T=int
    g(4);          // T=int par type de l'argument
    vector<int> v;
    //h(v);         //NON Pas de conversion implicite vers type générique
    h<int>(v);    // Conversion implicite vers type précisé
    h(span(v));   // Conversion explicite, T=int par type de l'argument
}
```

Exemple de classe générique

Paramètre taille est ***int constante à la compilation.***

Dit ***paramètre non-type.***

```
template <typename T, int taille>
class Tableau1 {
public:
    T& operator[] (int index) { return valeurs[index]; }
    const T& operator[] (int index) const { return valeurs[index]; }

    int size() const noexcept { return taille; }

private:
    T valeurs[taille] = {};
};
```

Utilisation:

Tableau1<int, 2> t;

t[0] = 4; t[1] = 2; **non const**

afficher(cout, &t[0], t.size()); **const**

const auto& ct = t; afficher(cout, &ct[0], ct.size());

Accesseurs const et non const;
pas encore de belle manière de définir les deux d'un coup.

taille et type T connus à la compilation.
{} initialise les éléments avec constructeur par défaut de T.

Définition des méthodes de classe générique

- Doivent être définies dans le .h
 - Car le corps doit être compilé pour chaque utilisation différente
 - .cpp sont compilés de manière indépendante
 - .h sont compilés avec chaque .cpp qui en fait #include
- Dans ou hors de la classe
 - Dans, comme d'habitude (voir exemple précédent)
 - Hors, doit répéter « template ... » :
`template <typename T, int taille>` Souvent dans la classe pour éviter ces répétitions
`T& Tableau1<T, taille>::operator[] (int index) {`
 `return valeurs[index];`
`}` Arguments du template

Définition de méthodes génériques d'une classe générique (dans la classe)

```

template <typename T, int taille>
class Tableau1 {
public:
    ...
    template <typename U>
    Tableau1<U, taille> convertirEn() const {
        Tableau1<U, taille> resultat;
        for (int i = 0; i < taille; ++i)
            resultat[i] = U(valeurs[i]);
        return resultat;
    }
    ...
};

Utilisation:

```

Tableau1<int, 2> t;
 $t[0] = 4; t[1] = 2;$

Tableau1 td = t.convertirEn<double>();
afficher(cout, &td[0], td.size());

Méthode template avec un autre type

Ici on utilise la même taille

Conversion explicite en type U
utilise un constructeur de U ou
opérateur de conversion

Type du template déterminé par la valeur (C++17)

Pourquoi le type est requis ici?

Définition de méthodes génériques d'une classe générique (hors de la classe)

```
template <typename T, int taille>
class Tableau1 {
public:
    ...
    template <typename U>
    Tableau1<U, taille> convertirEn() const;
    ...
};

template <typename T, int taille>
template <typename U>
Tableau1<U, taille> Tableau1<T, taille>::convertirEn() const {
    Tableau1<U, taille> resultat;
    for (int i = 0; i < taille; ++i)
        resultat[i] = U(valeurs[i]);
    return resultat;
}
```

Répéter template de la classe

Répéter template de la méthode

Ne pas oublier la classe avec arguments

Améliorer « afficher »

- Premier exemple était:

- ```
template <typename T>
void afficher(ostream& os, T valeurs[], int nValeurs)
```
- Doit passer l'adresse et taille (adresses contigües)
  - Permet tableau, array, vector, Tableau1
  - Pas list ou deque

- span regroupe l'adresse et taille

- ```
template <typename T>
void afficher2(ostream& os, span<const T> valeurs)
```
- Pas implicite:
 - `array<int,2> ar = {3, 2}; afficher2(cout, span<const int>(ar));`
 - `vector vec = {3, 2}; afficher2(cout, span<const int> (vec));`

Doit mettre le type pour que le span soit const

Compile une seule
version $T=int$

Améliorer « afficher2 » span

- On peut surcharger un template
 - Le plus constraint dont le type correspond a priorité
 - Ambigu si plusieurs aussi constraint correspondent
- Permettre « span » sans le type
 - Ajouter (en plus de afficher2 de la dernière page)
 - ```
template <typename T, size_t taille> void afficher2(ostream& os, span<T, taille> valeurs) {
 afficher2(os, span<const T>(valeurs));
}
```

    - Plus général, moins prioritaire
    - Convertit en const T
  - Pas implicite, mais type automatique:
    - `array<int,2> ar = {3, 2}; afficher2(cout, span(ar));`
    - `vector vec = {3, 2}; afficher2(cout, span(vec));`

2 versions de la ligne de conversion ci-dessus,  
une de l'affichage

# Plus générique

- Plus générique sans contrainte:

```
• template <typename T>
void afficher3(ostream& os, const T& valeurs) {
 os << "{ ";
 bool premiereFois = true;
 for (auto&& v : valeurs) {
 if (!premiereFois) os << ", ";
 os << v;
 premiereFois = false;
 }
 os << " }";
}
```

Tous les types sont compatibles avec ce paramètre

Erreur de compilation ici si T n'est pas itérable

- Fonctionne avec list, deque, ... pas besoin de conversion explicite

- int t[] = {3, 2};               afficher3(cout, t);
- array<int,2> ar = {3, 2};    afficher3(cout, ar);
- vector vec = {3, 2};           afficher3(cout, vec);
- list lis = {3, 2};             afficher3(cout, lis);
- //afficher3(cout, 4);

Compile une version par type et taille de tableau ou array.

erreur à la ligne du « for »

# Contraintes

C++20

(possible avant C++20 avec SFINAE pas matière au cours)

- Contraindre permet d'avoir
  - Une indication claire sur les types supportés
  - L'erreur sur la ligne qui fait l'appel, non dans la fonction
  - Plusieurs surcharges aussi contraintes mais disjointes
- Ajouter la contrainte à l'entête d'afficher3

#include &lt;ranges&gt;

Itérable au moins une fois du début à la fin

```
template <ranges::input_range T>
void afficher3(ostream& os, const T& valeurs) {
 ... même code qu'avant ...
}
```

Aussi générique que sans contrainte, si la contrainte correspond aux besoins du code ici

# Traits de type

⇒ Définis dans `<type_traits>`,  
`<iterator>`, `<ranges>` ... selon les traits

C++20

- Les traits de type (`type traits`) permettent
  - obtenir de l'information sur un type
  - modifier un type
- Un type est-il convertible en `span<const T>`?

- template <`typename T`> Trait de type qui donne le type des éléments d'un range  
`using EnSpanConst = span<const ranges::range_value_t<T>>;`

Un modèle d'alias définit une « fonction » donnant un type, pas une valeur.  
 Ex: `EnSpanConst<vector<int>>` donne `span<const int>`

- template <`typename T`> les nom...\_v sont des valeurs (souvent bool), ...\_t pour type  
`concept ConvertibleEnSpanConst = is_convertible_v<T, EnSpanConst<T>>;`

`concept nom = expression true/false pour dire quels types correspondent à ce concept`

vrai si le type est implicitement convertible en le second

# Contraintes composées

- Regrouper les `span<const T>` en une version de `afficher3`

Constraint par notre concept

```
template <ConvertibleEnSpanConst T>
void afficher4(ostream& os, const T& valeurs) {
 afficher3(os, EnSpanConst<T>(valeurs));
}
```

Un seul `afficher3` pour `int[N]`,  
`array<int,N>`, `vector<int>`

Conversion explicite dans le type résultant de `EnSpanConst`

- Supporter les autres types (séparés)

```
template <typename T>
requires ranges::input_range<T> && !ConvertibleEnSpanConst<T>
void afficher4(ostream& os, const T& valeurs) {
 afficher3(os, valeurs);
}
```

Sinon ambigu avec template ci-dessus

Un `afficher3` par type:  
`list<int>`, `deque<int>`

Contrainte syntaxe longue

## Fonction lambda ou anonyme

Expression lambda, fermeture / capture,  
std::function,  
**fonction d'ordre supérieur**

C++11

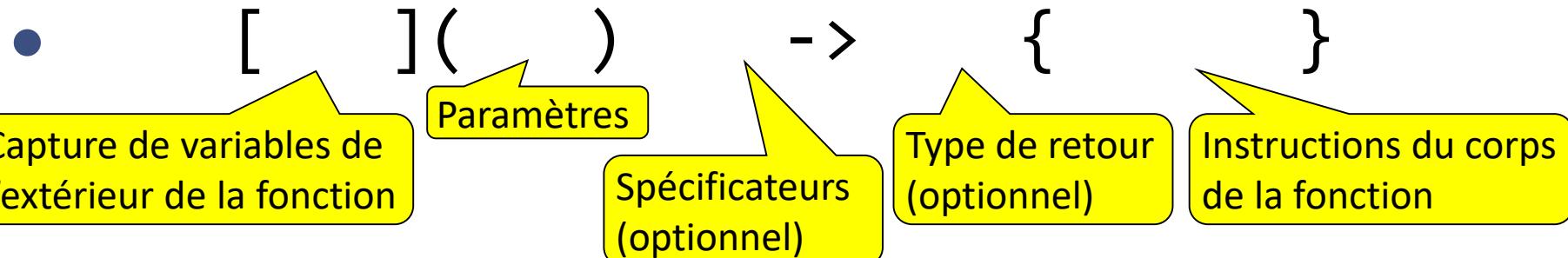
# Motivation

- On veut une fonction générale pour trouver dans un conteneur un élément qui répond à un critère
  - Trouver dans conteneur un élément où critère(element) est vrai
- On veut utiliser cette fonction pour trouver un élément plus grand que valeurMin
- On se demande
  - Comment écrire la fonction?
  - Comment faire l'appel?
  - Le critère sera une fonction?

Réponse possible:  
Une fonction d'ordre supérieur  
à laquelle une fonction lambda  
sera passée en paramètre

# Expression lambda

- Construit une *fermeture* (« closure »)
  - une fonction anonyme
  - pouvant capturer son environnement lexical
    - les variables à sa portée
  - écrite comme expression dans une autre fonction
- Objet appellable comme une fonction
  - Son type est « implementation defined »
  - Type *auto* pour la garder dans une variable



# Ma première fonction lambda

```
auto p = []() {
 cout << "une fonction lambda"
 << endl; };
p(); // Appel de la fonction
```

Pas de capture

Pas de paramètres

Pas de type de retour  
(pas de -> )

Type var = expression;  
Expression est la lambda  
qui crée l'objet fermeture

# Capture de variable externe

```
int a = 2;
auto p2 = [&]() { a = 10;
 return a*a; };

cout << "a " << a << " appel "
 << p2() << " a " << a << endl;
```

Affiche quoi?

Affiche:  
a 2 appel 100 a 10

Capture par référence

Sans type retour = déterminé par *return*

« a » est capturé puisque non déclaré dans la lambda

Quel type?

# Spécification des captures

- Capture de variables externes (existantes)
  - [] : pas de variable externe;
  - [=] : toute variable accédée est capturée par valeur
    - sauf l'objet *this* qui est toujours par référence en capture implicite (= le pointeur *this* capturé par valeur);
  - [&] : ... par référence;
  - [=, &x] : ... par valeur, sauf x par référence;
  - [&, x] : ... par référence, sauf x par valeur;
  - [x, &y] : x par valeur, y par référence, pas d'autres;
  - [&x = y] : x capture par référence y;
  - [x = y + 1] : x capture par valeur le résultat y + 1;
  - [this] : l'objet *this* par référence (pointeur par valeur);
  - [\*this] : l'objet *this* par valeur (copie);

C++14

C++17

# Mutable

- Spécificateur *mutable* permet de modifier les captures par valeur

```
int depart = 4;
auto prochain = [=]() mutable { return depart++; };
cout << prochain();
cout << prochain();
cout << prochain();
cout << depart;
```

Affiche quoi?

Affiche:

4564

# std::function (non auto)

⇒ Défini dans <functional>

- Un type connu pour une lambda (de type fixe)

```
Type « implementation defined »
```

```
auto a1 = [](int x, int y) -> int { return x + y; };
```

Exemple avec type de retour

---

```
function<int(int, int)>
```

```
function a2 = [](int x, int y) { return x + y; };
```

Lambda permet paramètres *auto* (*template implicite*)

```
auto a3 = [](auto x, auto y) { return x + y; };
```

function<auto...> pas possible

---

```
//function a4 = [](auto x, auto y) { return x + y; };
```

Conversion possible

```
function<int(int,int)> a5 = [](auto x, auto y) { return x + y; };
```

```
cout << a1(3, 2) << " " << a1(3, 2.5) << endl;
cout << a2(3, 2) << " " << a2(3, 2.5) << endl;
cout << a3(3, 2) << " " << a3(3, 2.5) << endl;
cout << a5(3, 2) << " " << a5(3, 2.5) << endl;
```

Affiche quoi?

5 5  
5 5  
5 5.5  
5 5

# Exemple std::function (où auto pas possible)

Similaire à *dict* Python

Type de clé

Type de valeur; toutes les valeurs doivent avoir le même type

```
map< string, function<double(double,double)> > operations = {
 {"+", [](double a, double b) { return a + b; } },
 {"-", [](double a, double b) { return a - b; } },
 {"*", [](double a, double b) { return a * b; } },
 {"/", [](double a, double b) { return a / b; } }
};
```

Clé

Valeur

```
cout << "Entrer deux valeurs et une opération à effectuer (+ - * /): ";
double x, y; string operation; cin >> x >> y >> operation;
cout << "La réponse est: " << operations[operation](x, y) << endl;
```

Affiche quoi pour « 3 2 / » ?

1.5

# Lambda énoncés quelconques...

```
int somme();
function<void(vector<int>)>
sommeVecteur = [&somme](vector<int> vecteur) {
 for (int& x : vecteur) {
 somme += x;
 }
};
vector<int> nombres = { 1, 2, 3, 4, 5 };
sommeVecteur(nombres);

cout << somme << endl;
```

Tout ce qu'on peut mettre dans une fonction peut être dans une lambda: for, if, déclaration d'une classe ...  
Contrairement à Python

# Fonction d'ordre supérieur

- Est une fonction qui a
  - une fonction comme paramètre, et/ou
  - retourne une fonction comme résultat.
- Ex: `trouver_si(valeurs, [](auto v) { return v > 4; });`
- Types possibles pour représenter la fonction
  - `std::function`
  - type générique (paramètre de *template*)
  - Pointeur de fonction si lambda sans capture ou fonction
    - (compatible avec C et ancien C++)

# Ma première fonction d'ordre supérieur

```
static const int indexe_pas_trouve = -1;

int trouver_si1(span<int> valeurs, const function<bool(int)>& critere) {
 int i = 0;
 for (auto&& v : valeurs) {
 if (critere(v))
 return i;
 ++i;
 }
 return indexe_pas_trouve;
}
```

Version sans *template*

Attention si on *template function*,  
pas de conversion implicite vers  
un type générique

Exemple d'appel:

```
vector<int> valeurs = { 3, 2, 7, 4 };
int position1 = trouver_si1(valeurs, [](auto v) { return v > 4; });
cout << position1 << ":" << valeurs[position1] << endl;
```

Lambda passée convertie en *function*

Affiche quoi?

# Fonction d'ordre supérieur

## complètement template

Types complètement génériques sans contrainte;  
noms comme indication de la sorte de type

```
template <typename Conteneur, typename PredicatUnaire>
int trouver_si2(const Conteneur& valeurs, const PredicatUnaire& critere)
{
 ...même corps...
}
```

Tout par const& comme on fait généralement pour des objets

std::find\_if et std::ranges::find\_if (C++20) sont similaires  
mais retournent un itérateur au lieu d'un int

Même exemple d'appel:

```
vector<int> valeurs = { 3, 2, 7, 4 };
int position2 = trouver_si2(valeurs, [](auto v) { return v > 4; });
cout << position2 << ":" << valeurs[position1] << endl;
```

Passe l'objet fermeture sans conversion;  
fonction possiblement recompilée chaque fois

# Fonction d'ordre supérieur

## qui retourne une fonction

```
template <typename T>
function<T()> incrementerParPas(T depart, T pas) {
 return [x=depart, pas]() mutable { return x += pas; };
}
```

T est connu avant de compiler le return,  
donc conversion implicite fonctionne

On conserve dans une variable x, car depart  
n'est pas un bon nom après modification

Exemple d'appel:

```
cout << incrementerParPas(2, 3)() << " ";
auto prochain = incrementerParPas(2.0, 0.5);
cout << prochain() << " ";
cout << prochain() << endl;
```

Peut appeler directement la fonction retornée

Affiche quoi?

Un langage porte son nom

# Curryfication

En l'honneur de Haskell Curry, mathématicien ayant posé les bases de la *programmation fonctionnelle*

- Fonction curryfiée prend un seul argument et retourne une fonction curryfiée sur le reste des arguments.

```
auto equationDroite(double m) {
 return [=](double b) {
 return [=](double x) {
 return m*x + b;
 };
 };
}
```

Bref, une lambda peut retourner une autre lambda

Exemple d'appel:

```
cout << equationDroite(0.5)(1)(3) << " ";
auto droite = equationDroite(0.5)(1);
cout << droite(0) << " " << droite(1) << " " << droite(2) << endl;
```

Affiche quoi?

# Utilisation dans une classe

```
class PetitesValeurs {
public:
 void ajouter(int i) { valeurs_.push_back(i); }
 void setMaximum(int maximum) { maximum_ = maximum; }
 bool valeursSontCorrectes(); // Méthode qui nous intéresse.
private:
 vector<int> valeurs_;
 int maximum_;
};

bool PetitesValeurs::valeursSontCorrectes() {
 return trouver_si2(valeurs_, [this](int v) { return v > maximum_; })
 == indexe_pas_trouve;
}
```

Exemple d'appel:

```
PetitesValeurs pv; pv.setMaximum(5);
pv.ajouter(4); cout << pv.valeursSontCorrectes();
pv.ajouter(6); cout << pv.valeursSontCorrectes();
```

Capture *this* pour *maximum\_*;  
[=] ou [&] aurait aussi capturé *this* implicitement

Affiche quoi?

## Méthodes virtuelles et classes abstraites

Héritage, redéfinition de méthodes, polymorphisme, virtual, « slicing », classes abstraites et pures, typeid

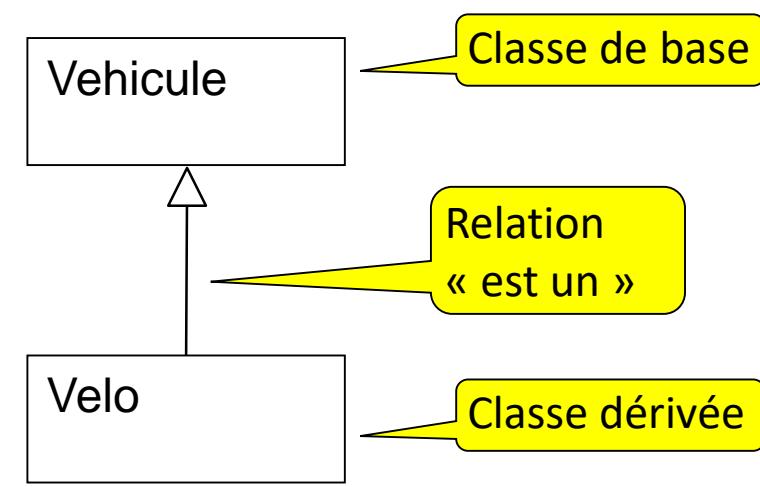
# Héritage (rappel)

- Une classe enfant/dérivée hérite des attributs et méthodes d'une classe parent/base.
  - Elle peut ajouter des méthodes et attributs.
  - Elle peut redéfinir des méthodes pour en changer le comportement par rapport à son parent.

- En C++

- `class Vehicule { ... };`
- `class Velo : public Vehicule { ... };`

Nous utiliserons toujours l'héritage public; l'héritage privé dit presque toujours « est implémenté en terme de » qui devrait être fait par composition si possible.



# Signification de l'héritage

- Attention, héritage = « est un »
- Ne pas confondre avec « est implémenté en terme de » ou « est composé de » = composition
- Le résultat paraît parfois équivalent, mais les concepts sont différents et la protection par encapsulation est différente
- Ex: Point et Cercle; Cercle: point (centre) et rayon
  - Cercle hérite de Point et ajoute un attribut rayon?
  - Question: un cercle est-il un point?
  - Ou Point hérite de Cercle et force le rayon à zéro?

## Signification de l'héritage (suite)

- Soit une classe Triangle: 3 points (sommets)
- On veut définir une classe Fleche, constituée d'un triangle et d'une droite perpendiculaire à un des côtés du triangle
- Fleche hérite de Triangle, et ajoute un attribut pour la droite?
- Est-ce raisonnable? Une flèche est-elle un triangle?
- → il faut décider du lien entre classes: héritage, composition ou agrégation

# Exemple de l'horloge

- Soit une classe Clock, qui permet d'obtenir l'heure locale, de deux façons: am/pm ou 24h (norme « militaire »)

```
class Clock
{
public:
 Clock(bool useMilitary); On n'a pas de constructeur par défaut
 string getLocation() const { return "Local"; }
 int getHours() const;
 int getMinutes() const;
 bool isMilitary() const;
private:
 bool isMilitary_; Unique attribut, de valeur spécifiée à la
 construction de l'objet; détermine le format
 de l'affichage de l'heure.
};

Utilisation:
Clock horloge1(true); Ex: « 23:45 »
Clock horloge2(false); Ex: « 11:45 »
```

## Exemple de l'horloge (suite)

- On aimerait une horloge pour l'heure d'une zone autre que locale

```
class TravelClock : public Clock
{
public:
 TravelClock(bool useMilitary, string location,
 int timeDifference);
 string getLocation() const { return location_; }
 int getHours() const;
private:
 string location_;
 int timeDifference_;
};
```

Ajoute deux attributs

Toujours pas de constructeur par défaut

Implémentation de méthode complètement différente de la classe de base. Dit redéfinition (« override »)

Méthode étendue de la classe de base pour ajouter le décalage horaire

# Constructeur de la classe dérivée

- Attention: la classe de base est construite avant le corps du constructeur de la classe dérivée

```
TravelClock::TravelClock(bool useMilitary, string location,
 int timeDifference) :
 Clock(useMilitary),
 location_(location),
 timeDifference_(timeDifference)
{}
```

Nécessaire pour spécifier comment construire la classe de base (sauf si construction par défaut)

L'initialisation des autres membres pourrait être dans le corps

# Appel des méthodes de la classe de base

- On peut appeler une méthode de la classe de base même si la classe dérivée la redéfinit

```
int TravelClock::getHours() const { int h = Clock::getHours() + timeDifference_; return modulo_positif(h, isMilitary() ? 24 : 12); }
```

Redéfinition qui utilise la méthode de la classe de base dite étendue

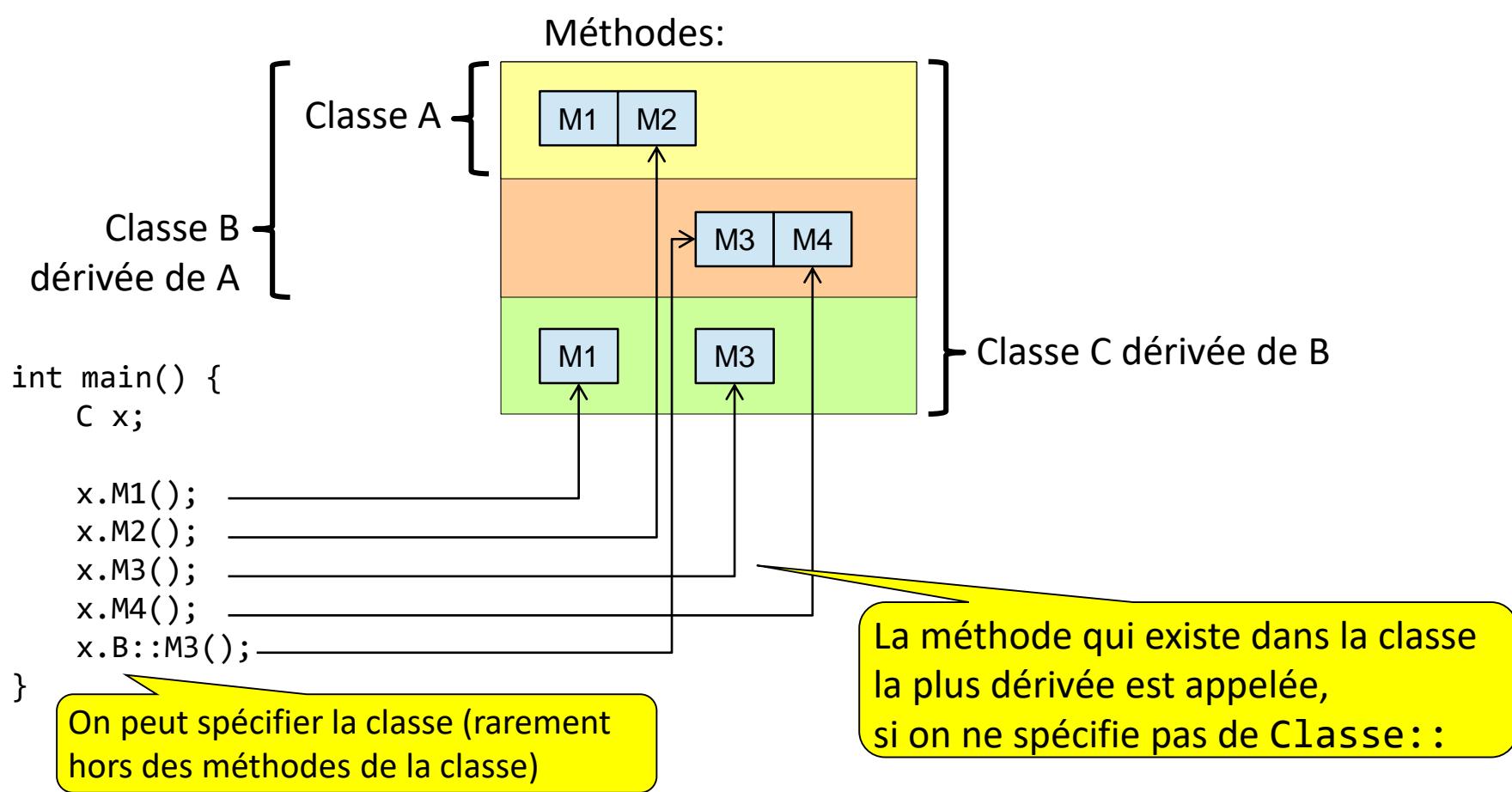
Spécifier la classe de la méthode pour appeler celle de la classe de base

```
int modulo_positif(int x, int n) { return (x % n) + (x < 0 ? n : 0); }
```

Méthode héritée, pas besoin d'indiquer la classe

Opérateur ternaire, comme Python:  
n if x < 0 else 0

# Appel d'une méthode



# Accès aux membres d'une classe de base

- **private** : accessible seulement dans la classe (et friend)
- **protected** : accessible à la classe et ses classes dérivées (et friend)
- En général, un attribut est toujours privé
- → pour qu'une classe dérivée puisse accéder à un attribut de la classe de base, on lui fournira une méthode d'accès protégée, s'il n'en existe pas déjà une publique

# Exemple accès protégé / privé

```
class Base
{
public: _____ Accessible à tous
 int getA() const;
 void setA(int x);
 int getB() const;
protected: _____ Accessible aux classes dérivées et friend
 void setB(int x);
 int getC() const;
 int setC(int x);
private:
 int a_, b_, c_; _____ Accessible juste dans cette classe
};
```

# Polymorphisme par héritage (rappel)

- Un objet peut avoir « plusieurs formes »
  - Avec une classe de base commune
  - Principe de substitution de Liskov dit en gros:
    - on peut utiliser un objet de classe dérivée comme un objet de la classe de base
- Supposons qu'on déclare trois horloges:
  - Clock c1(true);
  - TravelClock c2(true, "Paris", +6);
  - TravelClock c3(true, "Vancouver", -3);
  - Peut-on appeler une fonction `f(Clock c)` ?
  - Peut-on les mettre dans un `vector<Clock>` ?

TravelClock « est une » Clock,  
donc ce sont toutes des Clock

# Polymorphisme, premier test

- Fonction d'affichage comme on a l'habitude

Comme on passe les objets d'habitude

```
ostream& operator<< (ostream& os, const Clock& clock)
{
 return os << clock.getLocation() << ":" <<
 clock.getHours() << ":" << clock.getMinutes();
}
```

- On l'utilise

```
Clock c1(true);
TravelClock c2(true, "Paris", +6);
TravelClock c3(true, "Vancouver", -3);
cout << c1 << ", " << c2 << ", " << c3 << endl;
```

Compile mais affiche 3 fois « Local: 23:45 »  
??

# Polymorphisme, deuxième test : template

- Fonction d'affichage template (comme vu avant)

Concept C++20 dans <concepts>

```
template <derived_from<Clock> T>
ostream& operator<< (ostream& os, const T& clock)
{ ... même corps ... }
```

- On l'utilise

```
Clock c1(true);
TravelClock c2(true, "Paris", +6);
TravelClock c3(true, "Vancouver", -3);
cout << c1 << ", " << c2 << ", " << c3 << endl;
```

Ok: « Local: 23:45, Paris: 5:45, Vancouver: 20:45 »

Mais ça compile 2 versions, c'est du polymorphisme ad-hoc, pas par héritage

# Polymorphisme ad-hoc ou par héritage

- La version template fonctionne?

Pointeurs vers les horloges qui existent déjà, pour être certain que ce n'est pas un problème de constructeur de copie ; la conversion de pointeur ne modifie pas l'objet

```
vector<Clock*> horloges = { &c1, &c2, &c3 };
for (auto&& horloge : horloges) auto pour être certain qu'on a le bon type
 cout << *horloge << ", ";
cout << endl; Affiche 3 fois « Local: 23:45 »
```

- Pourquoi?

- Le type statique du pointeur est `Clock*`
- `<<` est appelé avec type statique `Clock&` Pourquoi & au lieu \* ?
- Dans `<<`, la version de `getHours` est choisie selon le type statique Dit aiguillage statique (ou liaison statique) « static dispatch »

# Polymorphisme par héritage, méthodes virtuelles

- On aimeraient dire
  - Le pointeur est `Clock*` mais j'aimerais appeler la méthode selon le « type réel » de l'objet pointé d'une classe possiblement dérivée de `Clock`

Son type dynamique, pouvant changer à l'exécution.

Dit aiguillage dynamique (ou liaison dynamique) « dynamic dispatch »

- Méthode virtuelle

```
class Clock
{ ...
 virtual string getLocation() const;
 virtual int getHours() const;
...
};
```

Un objet avec **au moins une méthode virtuelle** est dit polymorphe

On veut l'aiguillage dynamique

La boucle précédente donne maintenant ce qu'on aimeraient et on n'a pas besoin de << template

## Méthodes virtuelles (suite)

- *Attention:* si une méthode est virtuelle dans une classe, elle le sera automatiquement dans toutes ses classes dérivées
- Pour éviter toute confusion
  - Déclarer la méthode **override** dans la classe dérivée
    - S'assure que la méthode est virtuelle dans la classe de base
  - Vieux C++: déclarer **virtual** aussi dans les classes dérivées
- On n'a alors pas besoin d'aller consulter la classe de base pour savoir si une méthode est virtuelle

C++11

# Pourquoi pas tout « virtual »?

Similaire à Python

- Il y a un coût:

- Ajout d'un pointeur vers vtable
- Appel indirect par table de pointeurs de fonctions
- Le compilateur ne sais pas quelle fonction est utilisée
  - → ne peut pas optimiser l'appel avec « inlining »

Vu en INF1600

Enlever l'appel et le remplacer par ce que fait la fonction,  
ou directement son résultat

- « In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for » - Bjarne Stroustrup

(créateur du C++) La conception du langage est beaucoup basée sur ce principe

# Destructeur virtuel

- On a un tableau qui possède des horloges

{

```
array<unique_ptr<Clock>, 2> horloges = {
 make_unique<Clock>(true),
 make_unique<TravelClock>(true, "Paris, capitale de la France", +6)
};
```

Pointeurs intelligents, aucun « new » dans notre programme

...  
}

Fuite de mémoire (et *undefined behavior*); la string n'est pas désallouée

- Le destructeur de TravelClock n'est pas appelé

- Le destructeur n'est pas virtuel: choisi par le type statique Clock

On n'a pas déclaré de destructeur mais un est créé automatiquement si un attribut en a un

- Toute classe de base polymorphe devrait avoir un destructeur virtuel**

(ou protégé, dans des cas pas matière au cours)

# Exemple de l'horloge, résumé

```
class Clock
{
public:
 Clock(bool useMilitary);
 virtual ~Clock() = default;
 virtual string getLocation() const;
 virtual int getHours() const;
 int getMinutes() const;
 bool isMilitary() const;
private:
 bool isMilitary_;
};
```

Destructeur par défaut ici suffisant,  
mais doit spécifier **virtual** car la  
classe est polymorphe

Méthodes virtuelles pour  
polymorphisme  
(aiguillage dynamique)

```
class TravelClock : public Clock
{
public:
 TravelClock(bool useMilitary, string location,
 int timeDifference);
 string getLocation() const override;
 int getHours() const override;
private:
 string location_;
 int timeDifference_;
};
```

Redéfinition des méthodes virtuelles  
pour la classe dérivée; dire **override**  
pour lisibilité et s'assurer qu'il y a bien  
le **virtual** sur les bonnes méthodes de  
la classe de base

## « Object slicing »

- Peut-on mettre les objets dans le vector?

```
vector<Clock> horloges = { c1, c2, c3 };
for (auto&& horloge : horloges)
 cout << horloge << ", ";
cout << endl;
```

Affiche 3 fois « Local: 23:45 »

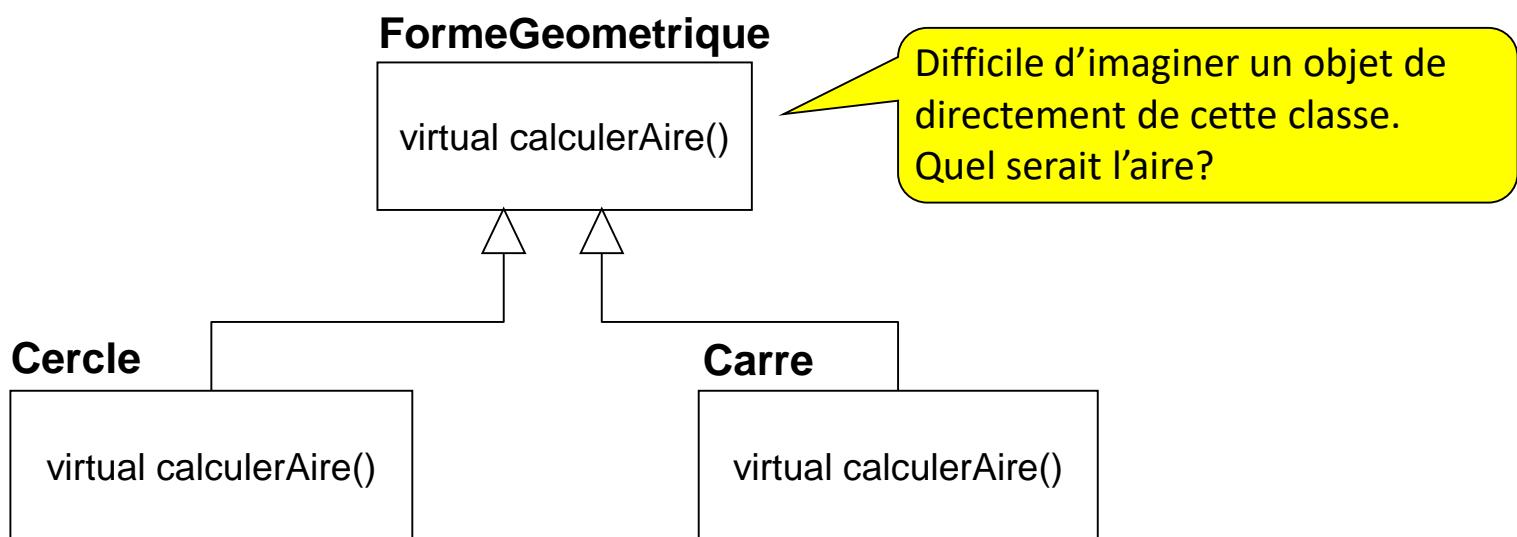
- Pourquoi?

- Copie dans des nouveaux objets de type Clock
  - Constructeur de copie de Clock
- Il « oublie » le reste qui n'entre pas dans Clock
- → ne peut pas contenir directement un objet de type dynamique différent du type statique
  - **Doit être pointeur** (intelligent) ou **référence**

Noter que **this** est un pointeur

Sinon pas de type dynamique différent,  
pas de liaison dynamique

# Classes abstraites (rappel)



- Une classe abstraite ne peut pas être instanciée
- Sert à établir des méthodes à définir dans les classes dérivées, et parfois des attributs communs

```

class FormeGeometrique {
public:
 virtual double calculateArea() = 0;
};

```

« =0 » sans définition, dit méthode virtuelle pure. Rend la classe abstraite.

Est-ce que tout est correct?

# Classe abstraite pure

- Classe abstraite: contient *au moins une* méthode virtuelle pure
  - Soit elle la déclare (avec « =0 »)
  - Soit elle en hérite et ne lui donne pas de définition
  - Aucun objet de cette classe ne peut être créé
- Classe abstraite pure: contient *uniquement* des méthodes virtuelles pures
  - Et un destructeur virtuel **=default** ou vide
  - Aucun attribut
  - Sert à définir une interface pour manipuler l'objet par polymorphisme (par référence/pointeur de ce type)

C++11

Destructeur ne peut pas être « =0 »

La déclaration de FormeGeometrique précédente répond-t-elle à ces critères?

# Obtention du type dynamique

- Peut-on compter les cercles?

```
span<unique_ptr<FormeGeometrique>> formes = ... ;
int compte = 0;
for (auto&& forme : formes)
 if (...?...) compte++;
```

- FormeGeometrique peut avoir une méthode virtuelle string obtenirType()
- Mais s'il sait quelle méthode appeler, il sait certainement le type?
- typeid(\*forme) == typeid(Cercle) Dans <typeinfo>
- Possible mais ne vérifie pas pour les classes dérivées de Cercle qui « sont des » cercles
- Mieux: dynamic\_cast<Cercle\*>(forme.get())

Vérifie l'héritage; voir prochain chapitre

## Conversion d'objet

« upcasting / downcasting », conversion statique / dynamique, explicit,  
prévenir le « slicing »

# Conversion classe dérivée → base

« upcasting »

Dans un diagramme d'héritage on dessine la classe de base en haut

- Un objet, pointeur/référence d'une classe dérivée peut être converti implicitement en un d'une classe de base

```
Clock c1(true);
TravelClock c2(true, "Paris", +6);
TravelClock c3(true, "Vancouver", -3);
```

Conversion implicite des objets: « slicing »

```
vector<Clock> horloges1 = { c1, c2, c3 };
```

```
vector<Clock*> horloges2 = { &c1, &c2, &c3 };
```

```
// TravelClock* x = horloges2[1];
```

NON Seulement vers une classe de base du type statique

Conversion implicite des pointeurs (ou référence) : objets intacts

Moins il y en a, mieux c'est

# Les « cast » selon le type statique

```
char lettre = 'A';
```

Le plus permissif  
(dangereux)

```
// En C:
```

```
int asciiLettre1 = (int)lettre;
```

```
// En C++:
```

```
int asciiLettre2 = static_cast<int>(lettre);
```

```
// En C++11:
```

```
int asciiLettre3 = int{lettre};
```

Empêche la conversion de  
pointeur en int, et vérifie  
l'héritage  
(moins dangereux)

```
char lettre2 = narrow_cast<char>(asciiLettre3);
```

```
char lettre3 = narrow<char>(asciiLettre3);
```

Vérifie qu'il n'y a pas de perte  
possible de données pour les type  
de base (sécuritaire)  
Mais « slicing » pour les objets

(GSL) Vérifie à l'exécution que  
la valeur n'est pas changée

(GSL) Dit que la  
restriction est voulue

# Conversion classe base → dérivée

## « downcasting »

- Jamais implicite

- Objet: peut définir un constructeur de conversion
- Pointeur/référence: doit faire un « cast »

```
TravelClock* x = static_cast<TravelClock*>(horloges[1]);
```

Vérifie uniquement que le type statique a un lien d'héritage

```
Clock c1(true);
```

```
TravelClock* y = static_cast<TravelClock*>(&c1);
```

*Undefined behavior* d'utiliser cet objet

```
TravelClock* z = dynamic_cast<TravelClock*>(horloges[1]);
```

Vérifie que le type dynamique est bien du type demandé (ou en dérive).  
L'objet doit être polymorphe.

C'est quoi « objet polymorphe »?

# « cast » selon le type dynamique: dynamic\_cast

- Vérifie à l'exécution le type dynamique
  - Utilise le RTTI (run-time type information)
    - Permet **typeid** selon le type dynamique
    - Se trouve par la vtable
  - L'objet doit être polymorphe pour avoir une vtable
- Pour convertir un pointeur ou référence

```
Classe_Dérivée* ptr_dérivée = Pointeur: nullptr si échoue
 dynamic_cast<Classe_Dérivée*>(ptr_base);
```

```
Classe_Dérivée& ref_dérivée = Référence: exception bad_cast si échoue
 dynamic_cast<Classe_Dérivée&>(ref_base);
```

# Exemple de dynamic\_cast

Secretary et Manager « sont des » Employee;

Secretary a une méthode getResponsability, Manager une méthode getBureau.

```
int main() {
 vector<Employee*> v;
 /* ... */
 int nbSecretary = 0;
 for (auto&& e : v) {
 if (auto secretary = dynamic_cast<Secretary*>(e)) {
 ++nbSecretary;
 cout << secretary->getResponsability() << endl;
 }
 else if (auto manager = dynamic_cast<Manager*>(e))
 cout << manager->getBureau() << endl;
 /* ... */
 }
}
```

Auto pour ne pas écrire  
Secretary\* deux fois  
(C++11)

Déclaration dans le  
« if » (C++17)

Changement dynamique de  
type; nul/faux si échoue

Appel de fonction de  
la classe dérivée

# Limiter les « cast », que faire?

- « upcast »
  - Utiliser la conversion implicite
- « downcast »
  - Préférer une méthode virtuelle si possible
    - et va avec la conception

Fonctionne aussi pour unique\_ptr et shared\_ptr

Aussi plus efficace que dynamic\_cast

Exemple précédent avec méthodes virtuelles isSecretary et print dans Employee:

```
int nbSecretary = 0;
for (auto&& e : v) {
 if (e->isSecretary())
 ++nbSecretary;
 e->print();
}
```

Fait du sens seulement si la classe de base peut savoir ce qu'est une secrétaire

Normal que les classes sachent comment s'afficher

# Restriction de conversion: constructeur explicit

```
class A {
public:
 A(int x) {}
};
class B {
public:
 explicit B(int x) {}
};
void TesterA(A t) { cout << "A\n"; }
void TesterB(B t) { cout << "B\n"; }
int main() {
 A a = 2; // Ok
 TesterA(2); // Ok
 B b1 = 2; // Non car explicit
 B b2(2); // Ok
 TesterB(2); // Non car explicit
 TesterB(B(2)); // Ok
}
```

Dit qu'il faut appeler  
explicitement ce constructeur

error C2664: 'void TesterB(B)' : impossible de convertir l'argument 1 de 'int' en 'B'  
note: Le constructeur pour class 'B' est déclaré 'explicit'

# Restriction de conversion:

## prévenir le « slicing »

Core Guidelines C.67

- Une classe polymorphe devrait empêcher la copie

Base(const Base&) = delete;

Base& operator= (const Base&) = delete;

Pas de constructeur de copie ni = de copie dans classe de base, se propage aux classes dérivées

- Comment copier si nécessaire?

Core Guidelines C.130

- Méthode virtuelle « clone() » pour copier selon le type dynamique
- Retourne un pointeur, soit

- `unique_ptr<Base>`
- `owner<Dérivée*>`

GSL

Toutes les définitions d'une méthode virtuelle doivent avoir les mêmes types, sauf le type de retour qui peut être « covariant », mais pas possible avec `unique_ptr`

# Exemple clone sans constructeur de copie: base

```

class D {
public: Constructeurs quelconques
 D(int x) : x_(x) {}
 D(const D&) = delete; Sans copie
 D& operator= (const D&) = delete;
 virtual ~D() = default; Destructeur virtuel car polymorphe

 virtual unique_ptr<D> clone() const {
 return make_unique<D>(CopySlice, *this);
 }
protected: Contre tag implicite {} Copie dans un nouveau pointeur du bon type
 struct CopySlice_t { explicit CopySlice_t() = default; };
 static constexpr CopySlice_t CopySlice{};
public: // mais protégé:
 D(CopySlice_t, const D& d) : x_(d.x_) {}
private:
 int x_;
};

```

« constructeur de copie » protégé par le tag;  
doit être public pour que make\_unique puisse l'appeler

Tag pour protéger la copie non voulue

# Exemple clone sans constructeur de copie: dérivée

```
class E : public D {
public:
 E(int x, int y) : D(x), y_(y) {}

 unique_ptr<D> clone() const override {
 return make_unique<E>(CopySlice, *this);
 }
};
```

Chaque classe redéfinit la méthode pour copier dans le bon type

```
public: // mais protégé:
 E(CopySlice_t, const E& e) : D(CopySlice, e), y_(e.y_) {}

private:
 int y_;
```

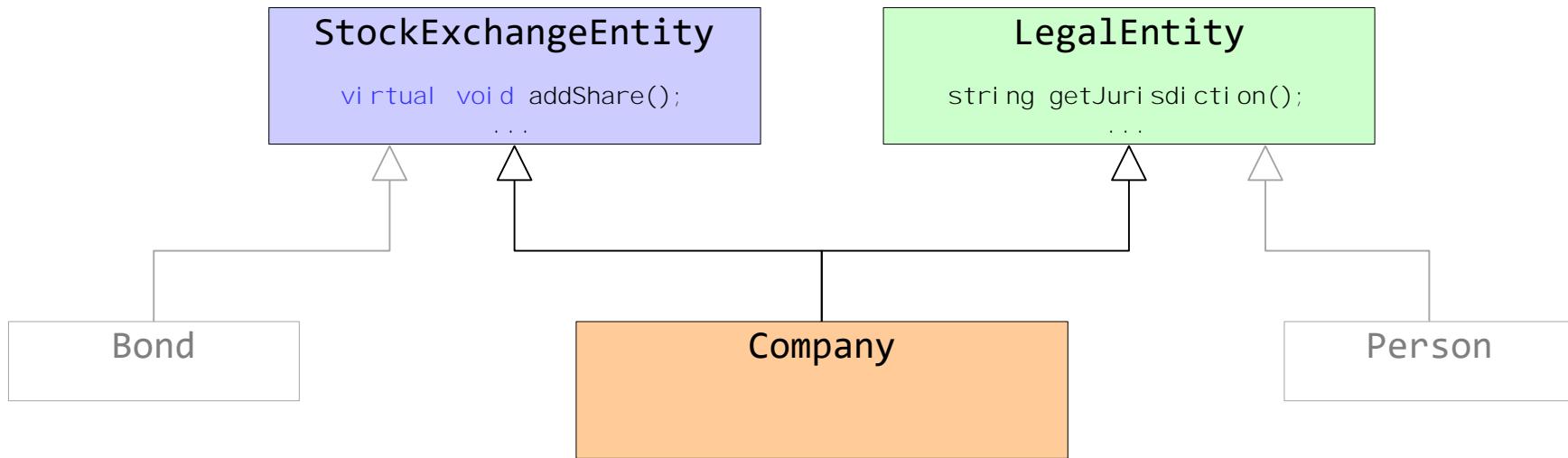
Et définit son « constructeur de copie » protégé en utilisant celui de la classe de base pour copier cette sous-partie de l'objet

## Héritage multiple

Et polymorphisme, ordre de construction,  
ambiguïté de nom, problème du diamant,  
attention, type vs interface

# Motivation

- Une compagnie peut être partagée en actions pouvant être en bourse (« stock exchange »)
- et est aussi une entité juridique (qui peut être menée en procès, ...) comme une personne
- Des obligations ou une personne humaine ne sont que l'un ou l'autre, donc ce sont des choses indépendantes



# Héritage multiple en C++

- En C++, on peut faire dériver une classe de plus d'une classe:

```
class Company
 : public StockExchangeEntity,
 public LegalEntity
{
 /* ... */
};
```

# Héritage multiple et polymorphisme

- Le polymorphisme fonctionne de la même manière avec l'héritage multiple
- La différence avec l'héritage simple est qu'un **même objet peut être pointé** (ou référé) par **deux pointeurs** (ou références) de **deux classes de base différentes**

# Héritage multiple et polymorphisme (exemple)

```
int main() {
 vector<StockExchangeEntity*> stock;
 vector<Legal Entity*> legal;
 /* ... */
 Company* c = new Company(/* ... */);
 /* ... */
 stock.push_back(c);
 legal.push_back(c);
 /* ... */
```

Le même objet mis dans deux vecteurs de types différents. (Les deux types sont parents mais n'ont pas d'héritage entre eux.)

```
int nbShares = dynamic_cast<StockExchangeEntity*>
(legal[0])->getNbShares();
}
```

Peut dynamic\_cast entre les bases.

Ne peut pas static\_cast entre les bases (erreur compilation).

# Héritage multiple et polymorphisme (exemple)

Polymorphisme par héritage: fonctions compilées une seule fois.

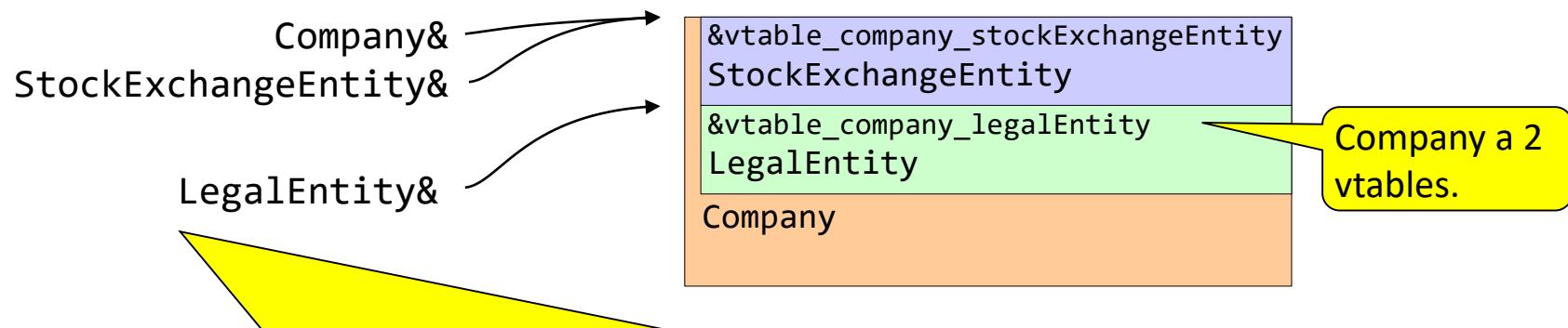
```
bool hasShares(const StockExchangeEnti ty& s) {
 return (s.getNbShares() > 0);
}
bool i sAtMontreal (const Legal Enti ty& l) {
 return (l.getJuri sdi ction() == "Montreal");
}
int mai n() {
 Company c/* ... */;
 if (hasShares(c)) {
 /* ... */
 }
 if (i sAtMontreal (c)) {
 /* ... */
 }
 /* ... */
}
```

Le même objet passé à des fonctions dont les paramètres sont de types différents.

# Héritage multiple et polymorphisme: le même objet a deux types sans lien?

- hasShare/isAtMontreal peuvent être compilées sans la définition des autres classes (fichiers .cpp/.hpp séparés).
- C'est donc le « upcast » (implicite) lors de l'appel qui doit s'assurer de passer la bonne chose, **sans modifier ou construire de nouvel objet, juste en ajustant l'adresse.**

```
class Company : public StockExchangeEntity, public LegalEntity { ... };
```



L'objet dérivé a la **même adresse que la première base**, mais pas que les autres bases; le « cast » change donc l'adresse. Une méthode virtuelle change aussi l'adresse de **this** (on ne voit pas les détails dans ce cours).

# Ordre de construction

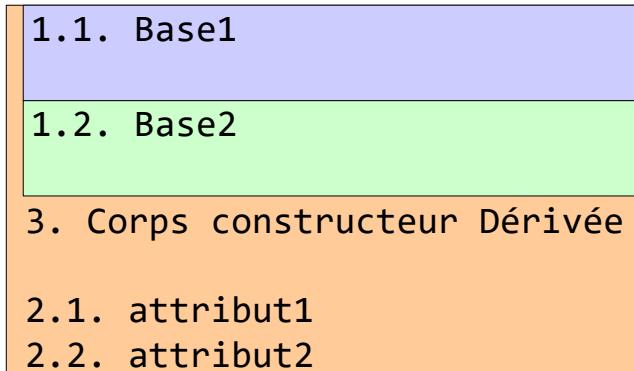
Les arguments de constructions sont évalués juste avant de commencer à construire le sous-objet concerné.

1. Bases dans l'ordre de la déclaration après class
2. Attributs dans l'ordre de leur déclaration
3. Corps du constructeur de la classe est exécuté

L'ordre est indépendant de l'ordre dans la liste d'initialisation du constructeur. **Pour éviter toute confusion, on écrit toujours la liste d'initialisation dans l'ordre des déclarations** (ordre de construction ci-dessus). Donc aussi indépendant de quel constructeur est utilisé.

```
class Dérivée : public Base1, public Base2 {
 ...
 Type1 attribut1;
 Type2 attribut2;
};
```

Chaque base et attribut est un sous-objet qui suivra aussi cet ordre de construction, récursivement.



# Ordre de construction, exemple à éviter

```
class MaClasse : public Base1, public Base2 {
```

```
public:
```

```
 MaClasse(int n) :
```

```
 nValeurs(n),
```

nValeurs n'est pas encore initialisé puisque sa construction est après

```
 Base2("bonjour"),
```

```
 valeurs(new int[nValeurs]),
```

```
 Base1(getTexteBase2())
```

```
 {}
```

```
private:
```

```
 int* valeurs;
```

Appel d'une méthode de Base2 qui n'est pas encore construite

```
 int nValeurs;
```

```
};
```

On pouvait déclarer nValeurs avant valeurs, pour l'initialiser dans le bon ordre, ou faire new int[n]

# Ordre de construction, exemple correct

```
class MaClasse : public Base1, public Base2 {
public:
 MaClasse(int n) : Liste dans l'ordre de construction, pour éviter la confusion
 Base1("bonjour"),
 Base2(getTexteBase1()), Base1 est bien construite lors de l'appel de la méthode
 nValeurs(n),
 valeurs(new int[nValeurs])
 {}
private:
 int nValeurs;
 int* valeurs;
};
```

nValeurs est bien initialisé (on a changé  
l'ordre de déclaration des attributs)

Si on voulait garder l'ordre des attributs, on pouvait mettre « new int[n] »  
ou mettre le « new int[nValeurs] » dans le corps du constructeur au lieu de  
la liste d'initialisation.

## Ordre d'appel des constructeurs, exemple

```
class A class B class C class D class E :
{ { { {
public: public: public: public:
 A(); B(); C(); D();
private: }; }; };
 B att_;
};

int main()
{
 E objet; On veut construire un objet de type E
}

Construit dans l'ordre:
A D C E() Le corps du constructeur de E sera exécuté en dernier

En souligné ce qu'il reste à voir comment ces objets sont construits
```

# Ordre d'appel des constructeurs, exemple

```

class A class B class C class D class E :
{ { { { {
public: public: public: public:
 A(); B(); C(); D();
private: }; }; };
 B att_;
}; 1.1
}; 1.2

int main()
{
 E objet;
}

Construct dans l'ordre:
A D C E()
 ↳ B A()

```

The code illustrates the constructor call sequence for class E. The classes are defined as follows:

- Class A:** Contains a public constructor A() and a private attribute B att\_.
- Class B:** Contains a public constructor B().
- Class C:** Contains a public constructor C().
- Class D:** Contains a public constructor D().
- Class E:** Contains a public constructor E(), a private attribute C att\_, and a public attribute B att\_. It also inherits from A and D.

Annotations with yellow boxes and numbers indicate the order of constructor calls:

- Annotation 1: Points to the constructor A() in class A.
- Annotation 2: Points to the constructor B() in class B.
- Annotation 3: Points to the constructor C() in class C.
- Annotation 4: Points to the constructor E() in class E.

A bracket labeled "Construct dans l'ordre:" groups the constructors A(), D(), C(), and E(). Below this bracket, a red bracket labeled "B A()" groups the constructors B() and A() from class A.

# Ordre d'appel des constructeurs, exemple

```

class A class B class C class D class E :
{ { { { {
public: public: public: public:
 A(); B(); C(); D();
private: }; }; };
 B att_; 1.1.1
}; 1.2
}; 1.1
}

int main()
{
 E objet;
}

Construct dans l'ordre:
A D C E()
 |
B A()
 |
B()

```

The code illustrates the constructor call sequence for class E. Annotations with yellow boxes and arrows indicate the order of constructor calls:

- 1**: Points to the constructor call in class E's definition: `public A,`
- 2**: Points to the constructor call in class D's definition: `public D`
- 3**: Points to the constructor call in class C's definition: `C()`
- 4**: Points to the constructor call in class B's definition: `B()`

A bracket labeled "Construct dans l'ordre:" groups the constructor calls in the order they are executed: A(), D(), C(), E(). Below this bracket, a hierarchical diagram shows the inheritance structure: A is the base class of D, which is the base class of E. B is a base class of A.

# Ordre d'appel des constructeurs, exemple

```

class A class B class C class D class E :
{ { { { {
public: public: public: public: public A,
 A(); } 1.2 B(); } 1.1.1 C(); } 2.1
private: }; }; };
 B att_}; } 1.1
};

int main()
{
 E objet;
}

Construct dans l'ordre:
A D
| |
B A()
| |
B()

```

The code illustrates the constructor call sequence for class E. It shows the inheritance path from A to E and the constructor calls for each class.

- Class A:** Contains a public constructor A() and a private attribute B att\_. Annotations: 1.1 (constructor call), 1.2 (constructor call).
- Class B:** Contains a public constructor B(). Annotation: 1.1.1 (constructor call).
- Class C:** Contains a public constructor C().
- Class D:** Contains a public constructor D(). Annotation: 2.1 (constructor call).
- Class E:** Contains a public constructor E() and a private attribute C att\_. Annotations: 1 (constructor call), 2 (constructor call), 3 (attribute declaration), 4 (constructor call).

A bracket labeled "Construct dans l'ordre:" groups the constructor calls: A(), D(), C(), E(). The inheritance hierarchy is shown below the bracket: A is the base class of D, which is the base class of E. B is a base class of A. B() is the constructor for B, A() is the constructor for A, D() is the constructor for D, and E() is the constructor for E.

# Ordre d'appel des constructeurs, exemple

```

class A class B class C class D class E :
{ { { { public A,
public: public: public: public:
 A(); } 1.2 B(); } 1.1.1 C(); } 3.1 D(); } 2.1
private: }; }; };
 B att_; }; }; };
}; }; }; };
}; }; }; };
int main()
{
 E objet;
}

Construct dans l'ordre:
A D C E()
 | | |
B A() D() C()
 | |
B()

```

1  
2  
3  
4

L'ordre de début de construction (évaluation des arguments s'il y en avait): E A B D C

Ordre de fin de construction = exécution des corps des constructeurs: B() A() D() C() E()

# Ordre de destruction

- Ordre inverse de la construction
  1. Corps du destructeur de la classe est exécuté
  2. Attributs dans l'ordre inverse de leur déclaration
  3. Bases dans l'ordre inverse de la déclaration après class

```
class Dérivée : public Base1, public Base2 {
 ...
 Type1 attribut1;
 Type2 attribut2;
};
```

Chaque base et attribut est un sous-objet qui suivra aussi cet ordre de destruction, récursivement.

L'ordre de début de destruction (exécution des destructeurs) est l'inverse de l'ordre de fin de construction (exécution des constructeurs)

3.2. Base1

3.1. Base2

1. Corps destructeur Dérivée

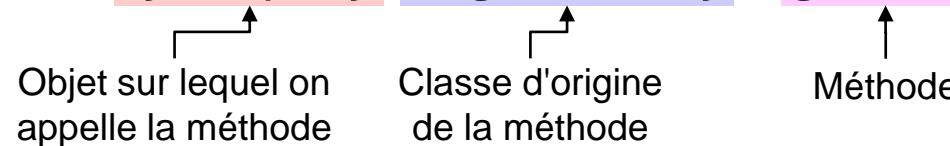
2.2. attribut1

2.1. attribut2

## Ambiguïté de nom

- Si StockExchangeEntity et LegalEntity ont toutes les deux une méthode int getId()
  - int id = myCompany.getId(); ————— Erreur, quelle méthode appeler?
- Une manière de régler le problème consiste à **indiquer explicitement la classe** de la méthode appelée :

```
int legalId = myCompany.LegalEntity::getId();
```



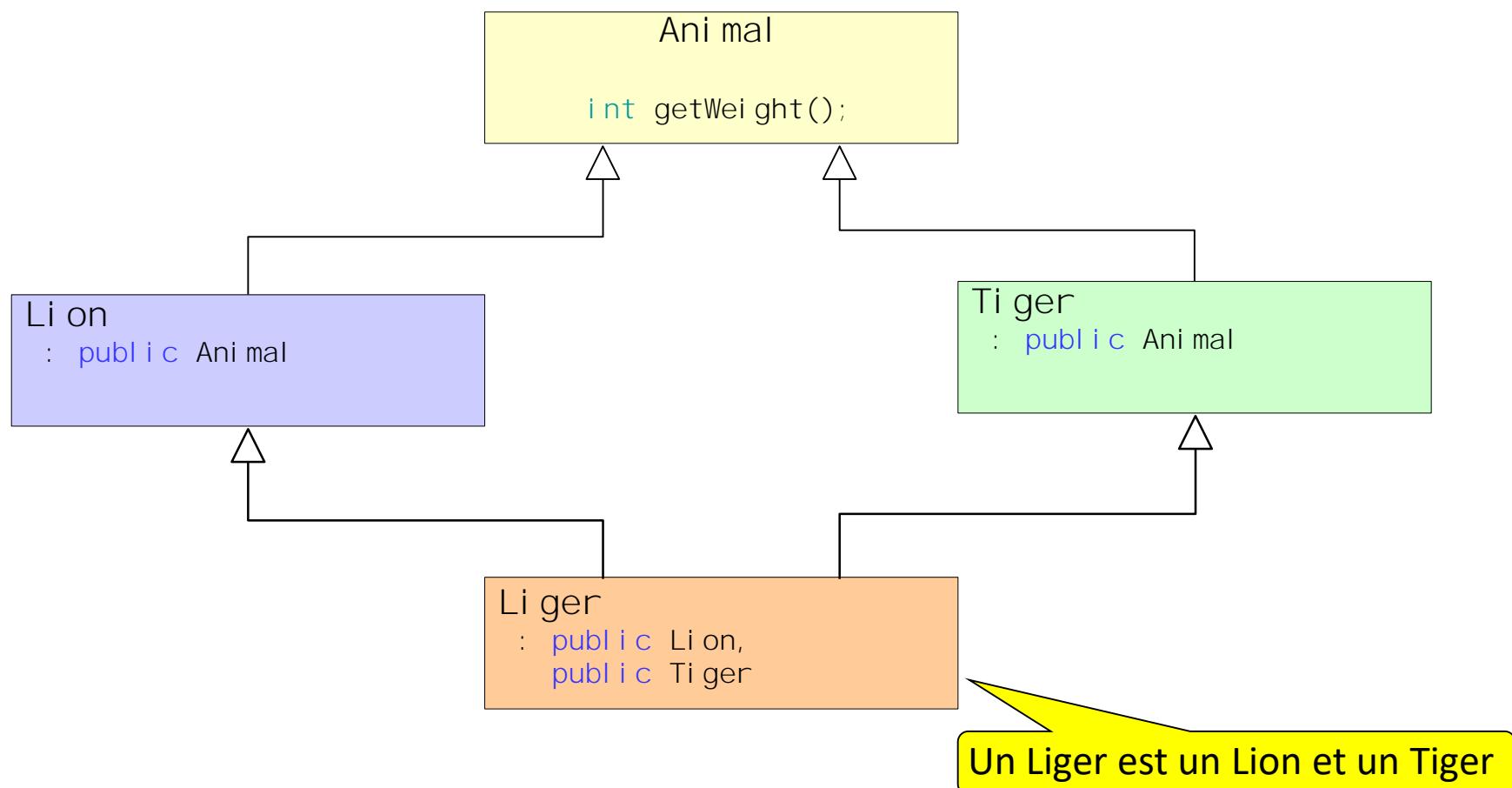
## Ambiguïté de nom (suite)

- **Une meilleure solution consiste à cacher l'ambiguïté et définir deux méthodes pour la classe Company:**

```
int Company::getLegalId() const {
 return LegalEntity::getId();
}

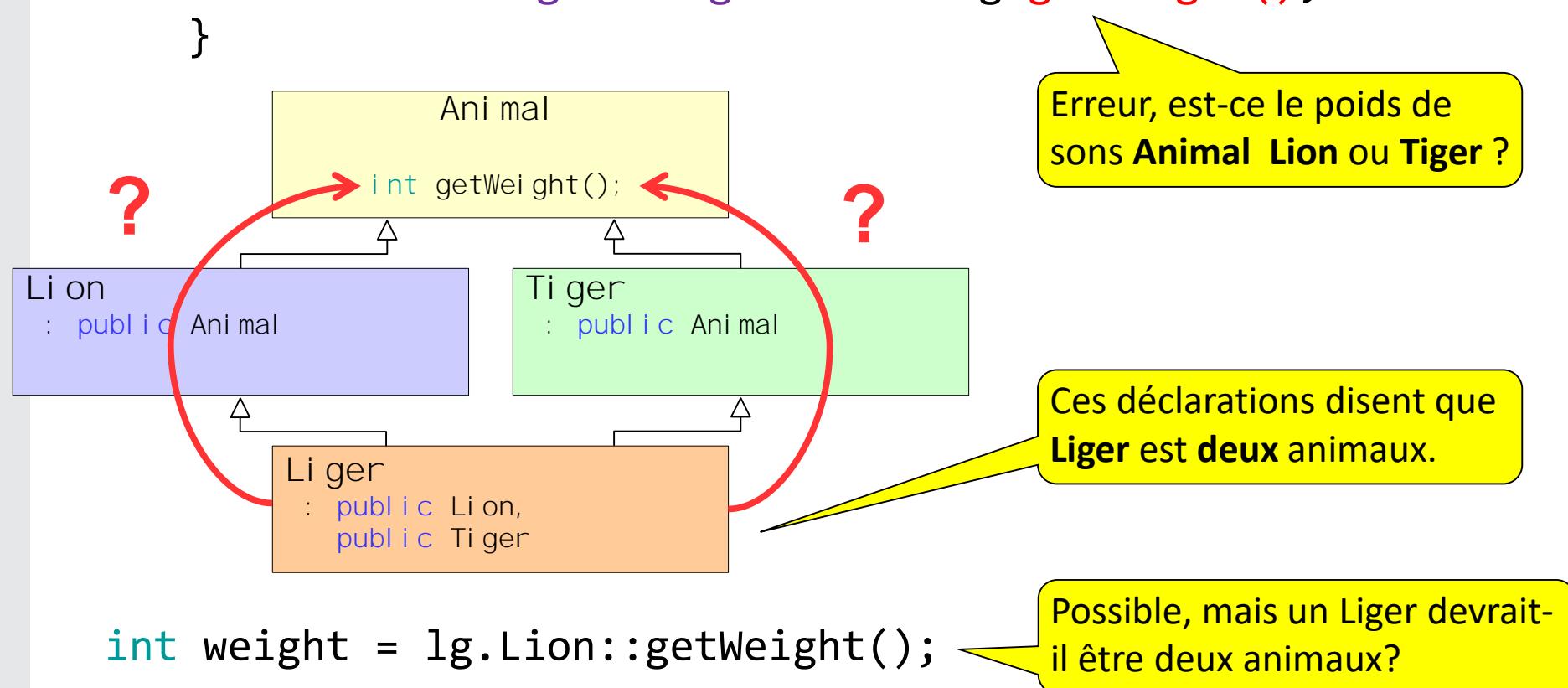
int Company::getStockId() const {
 return StockExchangeEntity::getId();
}
```

# Problème du diamant



# Problème du diamant (suite)

```
int main() {
 Liger lg;
 cout << "Liger weight: " << lg.getWeight();
}
```



# Problème du diamant (suite)

## héritage virtuel

- Un Liger n'étant pas deux animaux, on devrait utiliser l'héritage virtuel.
- Ça assure qu'un objet unique d'une classe multiplement dérivée existe: dans notre cas, un seul objet Animal, malgré que Tiger **et** Lion héritent de cette classe.
- Il suffit d'ajouter le mot clé **virtual** lorsqu'on hérite de la classe commune.

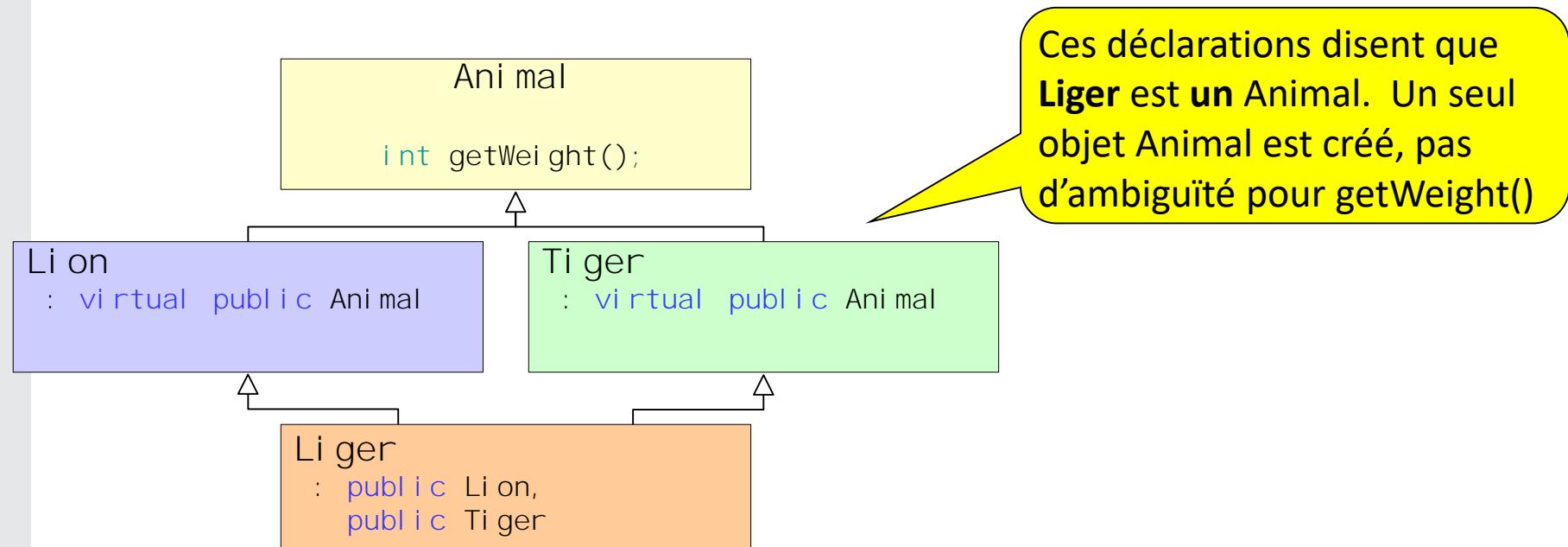
Il faut donc que cet héritage soit prévu par Lion et Tiger

```
class Lion : virtual public Animal /* ... */
class Tiger : virtual public Animal /* ... */
```

# Problème du diamant (suite)

## héritage virtuel

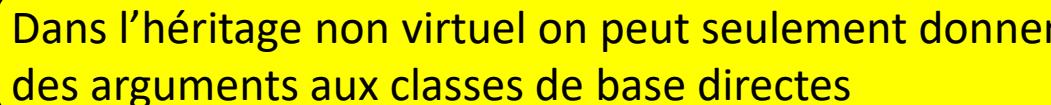
```
int main() {
 Liger lg;
 cout << "Liger weight: " << lg.getWeight();
}
```



# Problème du diamant (suite)

## héritage virtuel et construction

- Lion et Tiger ont une base Animal et peuvent spécifier des arguments de construction
  - Quand Animal est construit? Avec quels arguments?
- Toutes les bases virtuelles de la classe et ses ancêtres sont construites avant toutes les autres constructions des bases
- On doit spécifier les arguments des bases virtuelles dans toutes les classes dérivées
  - Liger::Liger(...) : Animal(...), Lion(...), Tiger(...) {}



Dans l'héritage non virtuel on peut seulement donner des arguments aux classes de base directes

# Attention avec l'héritage multiple

- D'autres problèmes peuvent être liés à l'héritage multiple
- L'héritage multiple de classes complètes peut compliquer la compréhension d'un code, et si un code n'est pas clair, il est difficile à partager et réutiliser
- Il ne faut pas utiliser inutilement l'héritage (multiple)
  - Est-ce que l'agrégation/composition serait préférable?
- L'héritage multiple d'interface est plus simple
  - Possiblement une classe de base complète + des interfaces

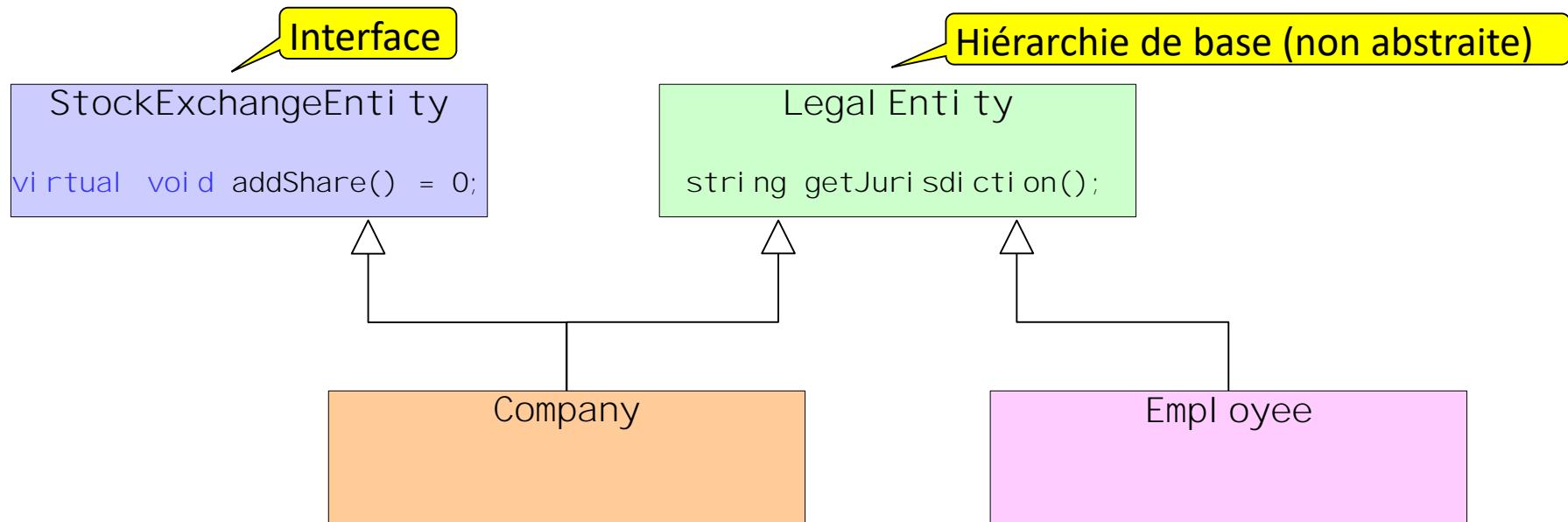
# Attention avec l'héritage multiple (suite)

Basé sur <https://isocpp.org/wiki/faq/multiple-inheritance>

- Utiliser l'héritage seulement si ça enlève un if / switch / map de fonctions
  - L'héritage est principalement pour l'aiguillage dynamique, qui est généralement préférable d'utiliser celui du compilateur que de tenter de le faire à la main
- Utiliser des interfaces (classes abstraites pures) au dessus du point d'héritage multiple
  - Pour ne pas hériter de code ou donnée par deux chemins

# Distinction entre type et interface

- Souvent une classe appartient à une seule hiérarchie de base
- à laquelle peuvent s'ajouter des interfaces



## Complexité et conteneur non contigu

Intro à la complexité, exemple liste liée,  
itérateur, boucles sur intervalles,  
types membres

# Motivation

- Imaginez un tableau contenant un certain nombre d'éléments
- On veut ajouter un élément après chaque élément qui répond à un critère
- Chaque ajout doit décaler tous les éléments après la position d'insertion
- Il s'agit d'une procédure coûteuse
  - Une insertion déplace moyenne ( $n$  éléments dans tableau)/2
  - Si nombre d'insertion proportionnel au nombre d'éléments, déplace total  $k n^2$  (pour une constante  $k$ ) dit  $O(n^2)$
- Solution possible: liste chainée/liée (« linked list »)

Suppose une distribution  
~uniforme des insertions

# Complexité algorithmique

Pour compréhension, il n'y a pas de limites ou preuves à faire dans ce cours.

- Notation  $O$  « ordre de complexité », d'un algorithme selon la taille de l'entrée
  - Notation asymptotique quand la taille tend vers  $\infty$
- Une définition:  $f(x)$  est dans  $O(g(x))$  ssi
  - il existe  $k$  et  $x_0$
  - tel que  $|f(x)| \leq k g(x)$  pour tout  $x \geq x_0$
- En langue parlée « cet algorithme est  $O$  de  $n$  »
  - Sous entend « complexité de temps » et  $n$  est le nombre d'éléments en entrée
  - Sinon doit spécifier « temps » ou « mémoire » et ce que représente la variable

# Complexité algorithmique (suite)

- Noter:
  - L'ordre de complexité est indépendant de la vitesse de l'ordinateur
    - Un algorithme deux fois plus vite a la même complexité
  - Plus complexe implique plus lent/gros quand  $n$  grand
    - Peut-être pas si  $n$  petit
  - $O$  d'une somme est  $O$  du plus gros terme
    - C'est-à-dire  $g(n) + f(n)$  est  $O(g(n))$  si  $f(n)$  est  $O(g(n))$
  - Donc tout polynôme en  $n$  de degré  $d$  est  $O(n^d)$ 
    - Ex.:  $n^2 + n$  est  $O(n^2)$  (car  $n^2 + n \leq 2n^2$  si  $n \geq 1$ )
  - Pour toute base  $b$  constante  $\log_b n$  est  $O(\log n)$

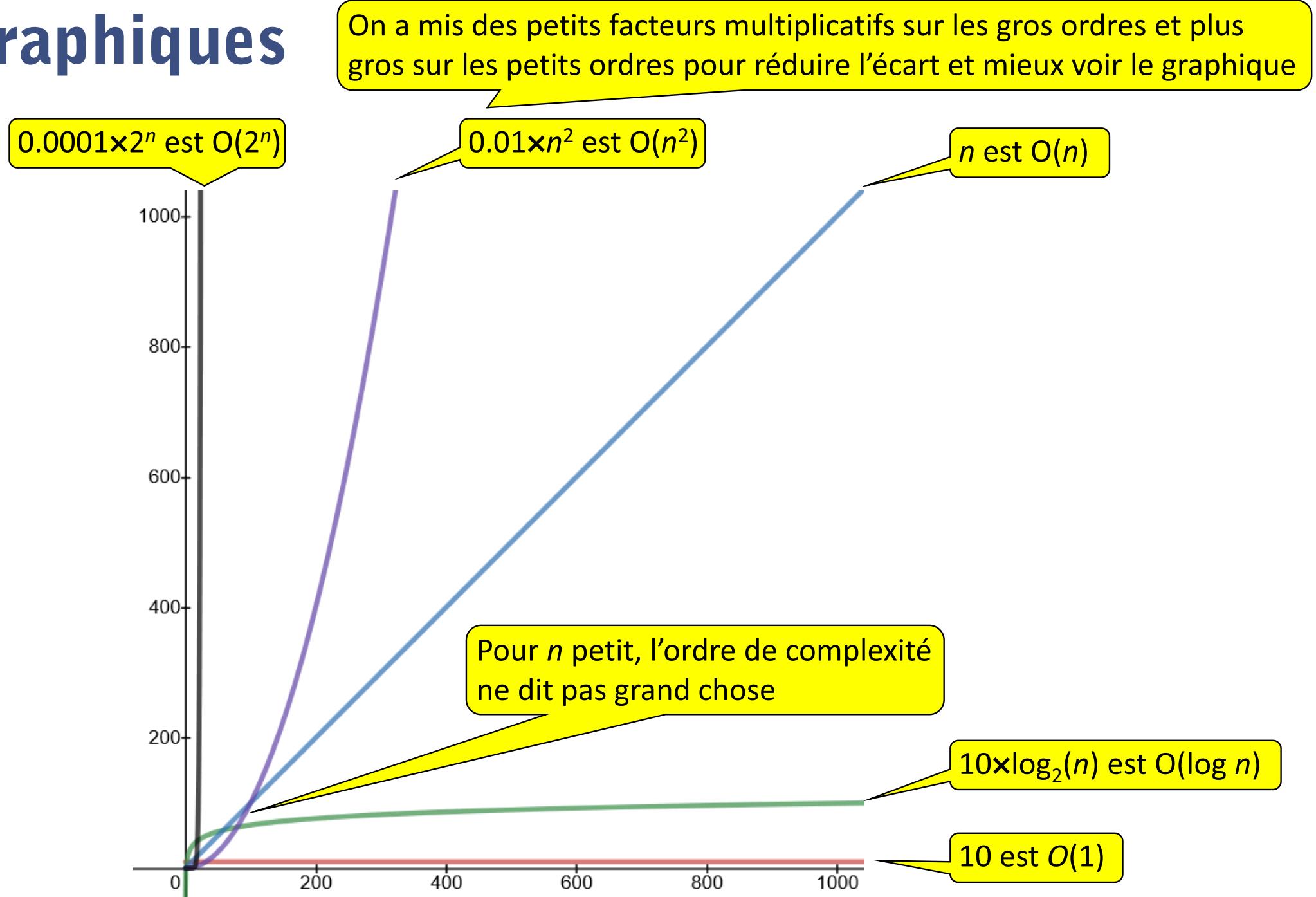
# Complexité algorithmique (suite)

## cas en ordre croissant de complexité

- Constant :  $O(1)$ 
  - Opération de base entre types de base (entier, réel, caractère, pointeur; *string* seulement si les tailles sont bornées d'avance)
- Logarithmique :  $O(\log n)$ 
  - Tant qu'il y a plus d'un élément dans le tableau considéré, tester en  $O(1)$  quelle moitié du tableau on doit continuer à considérer
- Linéaire :  $O(n)$ 
  - Pour chacun des  $n$  éléments du tableau, faire op. en  $O(1)$ 
    - $O(n)$  même si passe seulement une fraction des éléments, si la fraction ne tend pas vers 0 quand  $n \rightarrow \infty$
- Quadratique :  $O(n^2)$ 
  - Exemple précédent de  $O(n)$  insertions dans un tableau de taille  $n$ .
- Exponentiel :  $O(c^n)$ 
  - `int fib(int n) { if (n <= 1) return n; else return fib(n - 1) + fib(n - 2); }`
  - Est une catégorie, pas un ordre particulier, car  $3^n$  n'est pas dans  $O(2^n)$  ...

# Complexité algorithmique (suite)

## graphiques

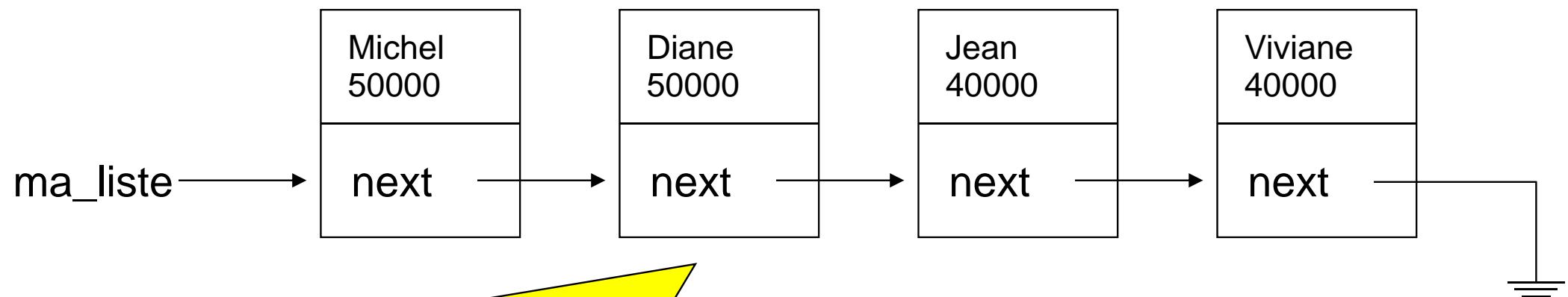


# Introduction aux conteneurs non contigus:

## Liste liée simple

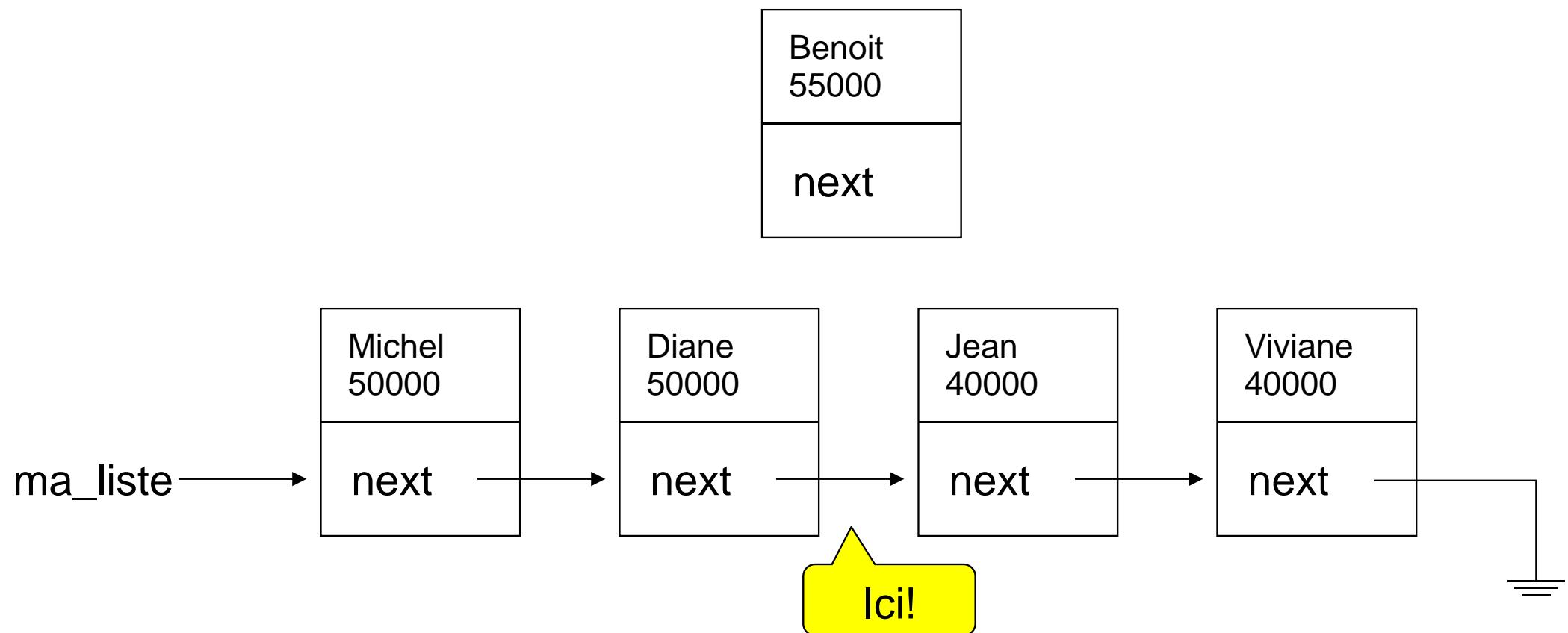
Contrairement à tableau/array/vector

- Les éléments ne se suivent pas en mémoire
- On conserve un pointeur vers le premier élément
- Chaque élément a un pointeur vers le suivant



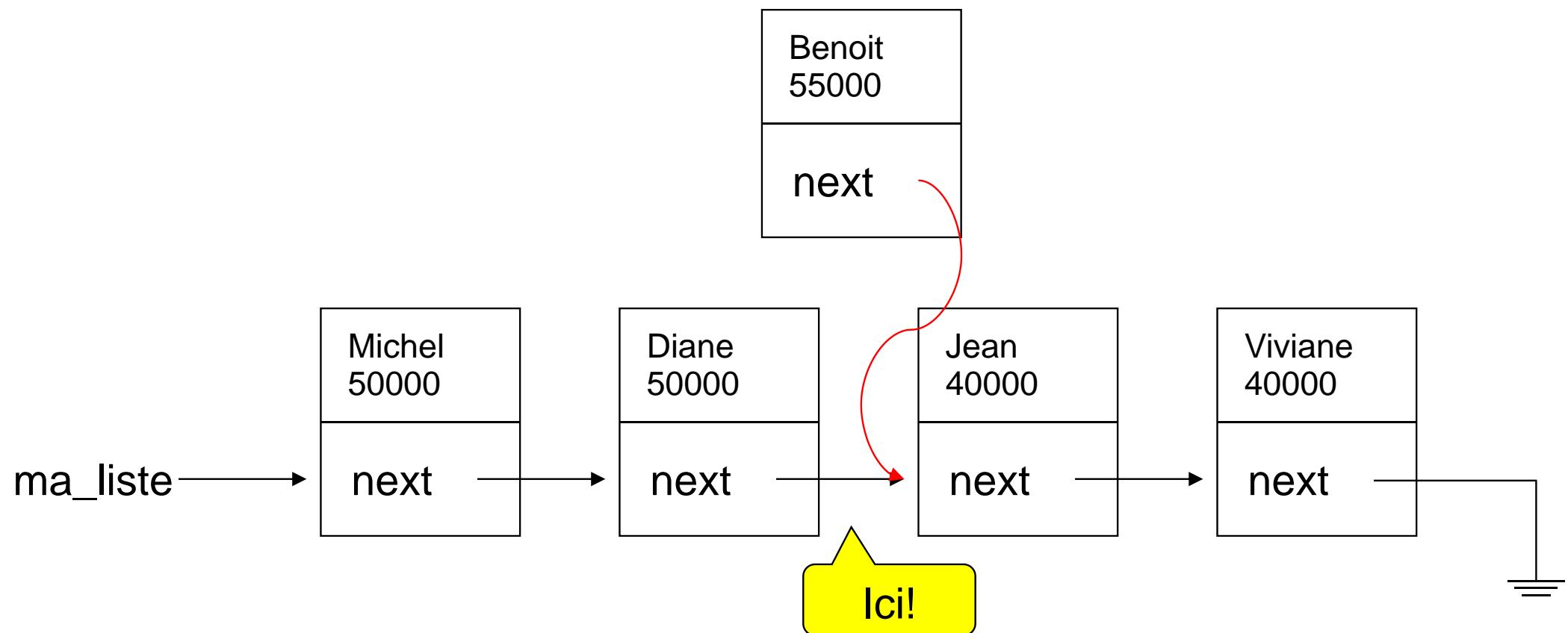
Montré en ordre de la liste, mais Diane pourrait être avant ou après Michel en mémoire...

# Ajout d'un élément



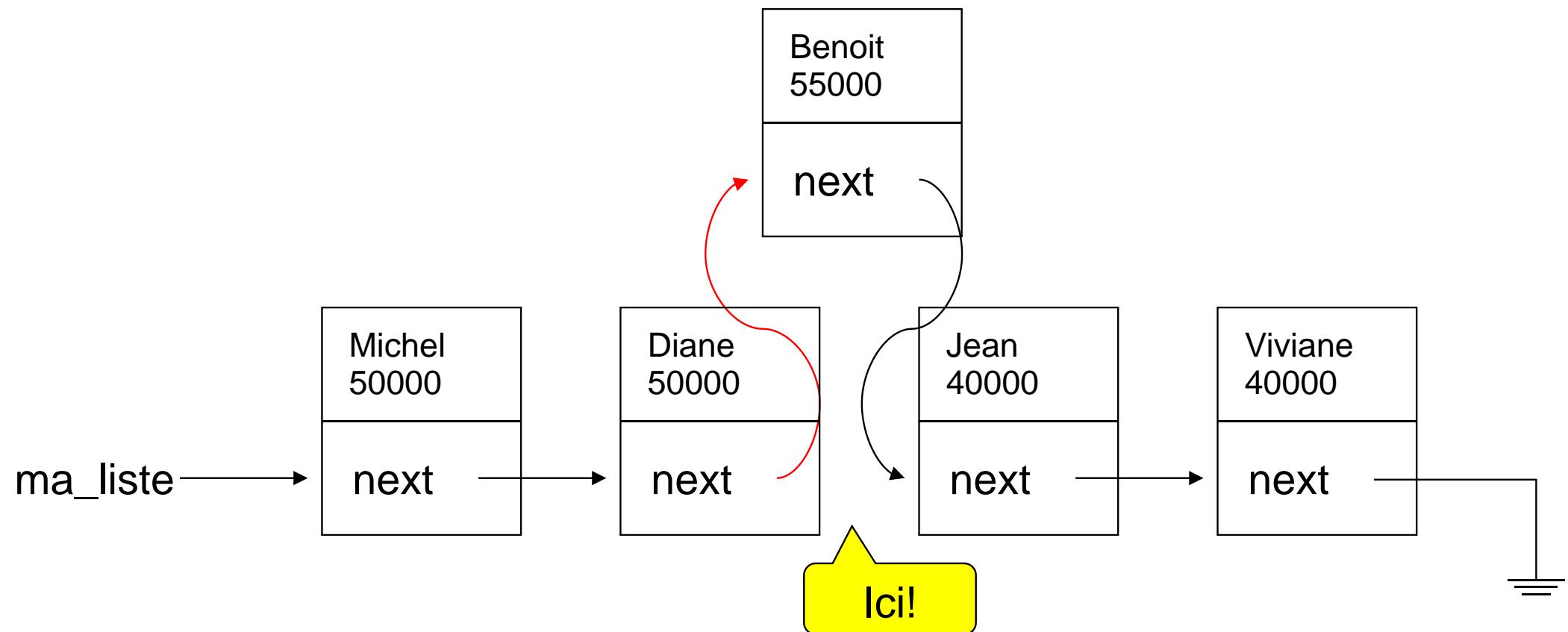
# Ajout d'un élément (suite)

## suivant du nouveau



# Ajout d'un élément (suite)

## suivant du précédent

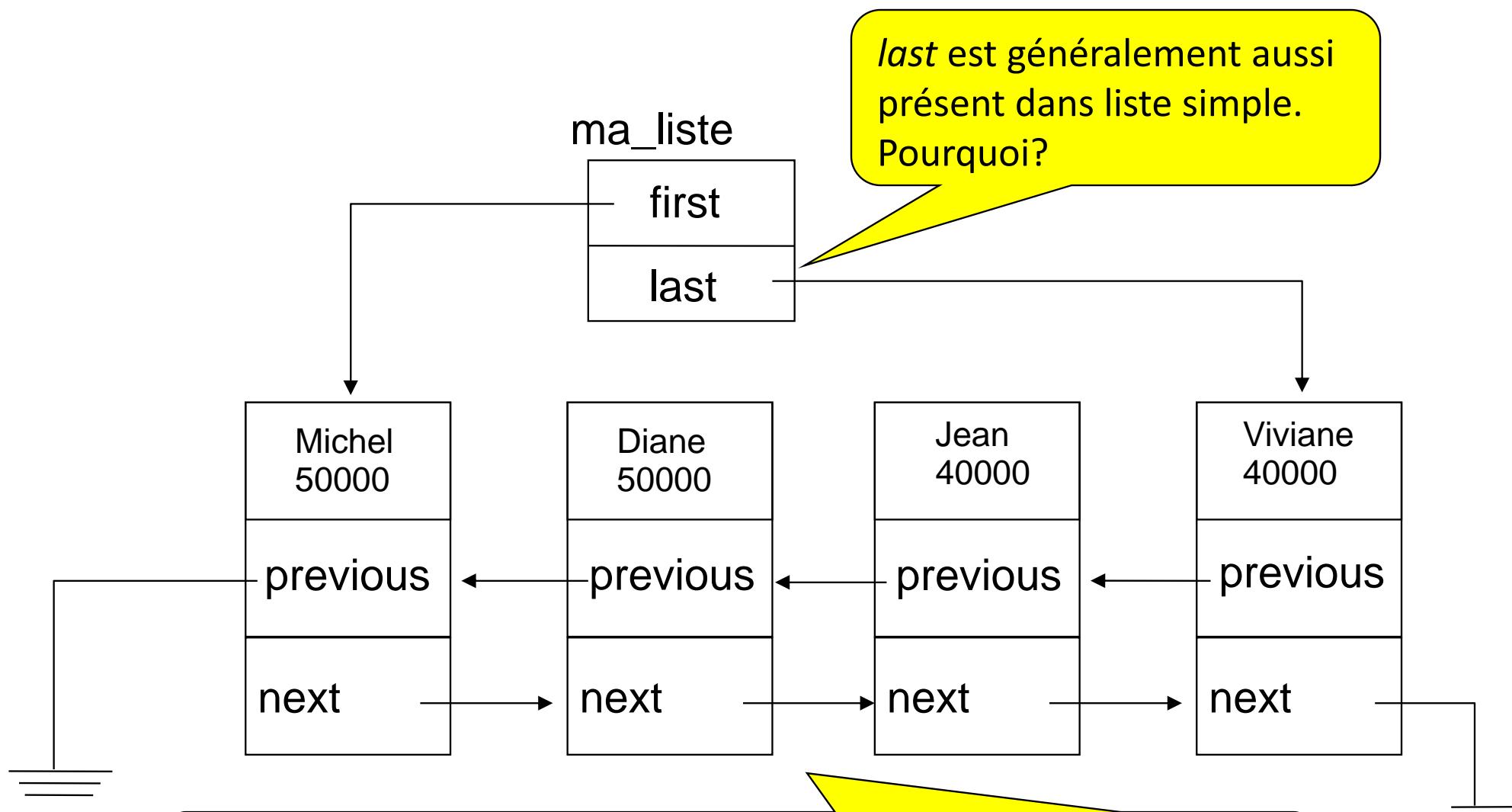


# Ajout d'un élément

- Peu coûteux
  - Seulement deux affectations de pointeurs
  - Temps constant  $O(1)$
- Anciens éléments pas déplacés
  - Pointeurs vers ces éléments restent valides
- **Mais:** pour trouver le  $i^{\text{ème}}$  élément il faut parcourir la liste à partir du début
  - Temps linéaire  $O(i) = O(n)$  si  $i$  proportionnel à  $n$
  - ( alors que  $\text{tableau}[i]$  est  $O(1)$  )

Pour cette raison, `std::list` ne définit pas l'opérateur `[]`

# Liste à liens doubles



*last* est généralement aussi présent dans liste simple.  
Pourquoi?

Liens doubles permettent de parcourir la liste dans les deux sens.  
**std::list** est à liens doubles, **std::forward\_list** est à liens simples.

# Implémentation de listes simples

- Pour implémenter les listes, il faut avoir trois classes:
  - List, pour représenter la liste
  - Node, pour représenter un élément dans la liste
    - Cette classe devrait être privée (comme dans std:: )
    - L'utilisateur de la liste n'a pas à savoir ce qu'est Node
  - Iterator, pour implémenter le parcours de la liste

# Classe Node - définition

Bleu pour liens doubles

```
class Node {
public:
 Node(const Employee& e);
private:
 Node* next_ = past_end;
 Node* previous_ = past_end;
 Employee employee_;

 inline static constexpr Node* past_end = nullptr;
 friend class List;
 friend class Iterator;
};
```

Tout devrait être générique mais on garde l'exemple « simple ».

Pointeurs par défaut.

Constante pour indiquer la fin de liste, utilise simplement *nullptr*.

*friend class* permet à List et Iterator de manipuler directement les attributs privés d'un Node.

Note: std:: n'a pas de classe noeud publique/accessible.

## Remarques sur friend

- List et Iterator sont intimement liées à Node
- Node n'est qu'une implémentation des données qui restera cachée à l'utilisateur de la liste
- Alors: *friend* ne brise pas l'encapsulation ici
- Peut implémenter sans utiliser *friend*
  - avec des méthodes accesseurs,
  - ou tout public dans Node, mais Node pas public
    - Actuellement dans MSVC, gcc (libstdc++), clang (libc++)

# Itérateur

Ressemble à Python

- Itérateur: généralisation d'un indice de tableau
    - aux structures possiblement sans indexation [i] facile
    - «pointeur intelligent» pour parcourir des données
  - Il «pointe» vers un élément de la liste
  - De base, un itérateur permet: `for (i = début; i != fin; ++i)`
    - Accéder à l'élément pointé
    - Aller à l'élément suivant `++i`
    - Comparer si deux itérateurs pointent vers le même élément `i!=fin`
  - Certains itérateurs permettent aussi:
    - Reculer
    - Avancer/reculer de  $n$  éléments
    - Savoir si un itérateur est avant/après un autre dans les éléments
    - Savoir le nombre d'éléments entre deux itérateurs
- Un pointeur brut permet tout ça si le conteneur est contigu; c'est un itérateur
- Pas propriétaire.
- Pourquoi pas  $i < \text{fin}$  ?

# Parcours d'une liste

Un itérateur de début et de fin définissent un intervalle dans un conteneur, c'est la base des **ranges C++20**

- Commence avec *pos* = itérateur qui pointe le premier élément (méthode *begin*)
- On définit *fin* = itérateur qui pointe **après** le dernier élément (méthode *end*)
- Jusqu'à ce que *pos* soit *fin* (operator!=):
  - Vérifier si l'item pointé par *pos* est celui recherché (operator\*, ou aussi operator-> dans std::list)
  - Si oui, on arrête
  - Sinon on fait pointer *pos* à l'item suivant (operator++)

# Classe Iterator – définition

Bleu pour liens doubles

```
class Iterator {
public:
 Iterator(Node* position = Node::past_end);
```

position par défaut.

```
Employee& operator*();
Iterator& operator++();
Iterator& operator--();
```

Opérateur de déréférence

C++20, définit == et != qui  
comparent tous les membres.  
Pourquoi les deux *const*?

```
bool operator== (const Iterator& b) const = default;
private:
 Node* position_;

 friend class List;
};
```

Si on appelle operator++() alors  
qu'on est à la fin de la liste, **position**  
aura alors la valeur **past\_end**.

# Classe Iterator – implémentation

```
Iterator& Iterator::operator++ () {
 Expects(position_ != Node::past_end);
 position_ = position_->next_;
 return *this;
}
```

gsl: vérification de précondition;  
similaire à **assert**

Retourne l'itérateur lui-même,  
permet ++ ++itér

```
Employee& Iterator::operator* () {
 Expects(position_ != Node::past_end);
 return position_->employee_;
}
```

On peut accéder directement  
l'attribut privé car *friend* de Node

Accesseur par référence non  
const, pour pouvoir modifier le  
contenu des éléments de la liste.

# Classe List - Définition

Vert pour liens simples

```

class List {
public:
 ~List(); Il faudrait aussi copie/move, mais on garde l'exemple « simple »
 void push_back(const Employee& s); Désalloue les Node.
 Pour ajouter à la fin.
 Iterator insert_after(Iterator pos, const Employee& s);
 Iterator erase_after(Iterator pos); Pourquoi after ?
 Iterator before_begin(); Pourquoi before ?
 Iterator begin(); // itér. qui pointe premier item
 Iterator end(); // itér. pointe après dernier item
 bool empty() const; // vrai si vide
private:
 Node* first_ = Node::past_end;
 Node* last_ = Node::past_end; Liste initialement vide
};
```

# Exemple de parcours de liste

```
Iterator find(List& list, const string& name) {
 Iterator fin = list.end();
 for (Iterator pos = list.begin(); pos != fin; pos.next())
 if ((*pos).getName() == name)
 return pos;
 return fin;
}
```

```
int main() {
 List staff;
 staff.push_back(Employee("Michel", 5000));
 staff.push_back(Employee("Roberta", 5000));
 staff.push_back(Employee("Claudia", 45000));
 staff.push_back(Employee("Mohamed", 43000));
```

Remplit la liste.

```
 Iterator claudia = find(staff, "Claudia");
 if (claudia != staff.end())
 ...
```

Appelle la fonction.

# Exemple de parcours de liste

Pourquoi pas const ici?

```
Iterator find(List& list, const string& name) {
 Iterator fin = list.end();
 for (Iterator pos = list.begin(); pos != fin; pos.next())
 if ((*pos).getName() == name)
 return pos;
 return fin;
}

int main() {
 List staff;
 staff.push_back(Employee("Michel",5000));
 staff.push_back(Employee("Roberta",5000));
 staff.push_back(Employee("Claudia",45000));
 staff.push_back(Employee("Mohamed",43000));

 Iterator claudia = find(staff, "Claudia");
 if (claudia != staff.end())
 ...
```

Initialise un itérateur pointant le premier élément de la liste et un autre pointant après le dernier élément.

# Exemple de parcours de liste

```
Iterator find(List& list, const string& name) {
 Iterator fin = list.end();
 for (Iterator pos = list.begin(); pos != fin; pos.next())
 if ((*pos).getName() == name)
 return pos;
 return fin;
}
```

Pourquoi (...). et non ->

On parcourt la liste jusqu'à ce qu'on trouve l'élément recherché.  
L'algorithme est  $O$  de quoi?

```
int main() {
 List staff;
 staff.push_back(Employee("Michel",5000));
 staff.push_back(Employee("Roberta",5000));
 staff.push_back(Employee("Claudia",45000));
 staff.push_back(Employee("Mohamed",43000));
```

```
 Iterator claudia = find(staff, "Claudia");
 if (claudia != staff.end())
 ...
```

# Exemple de parcours de liste

```
Iterator find(List& list, const string& name) {
 Iterator fin = list.end();
 for (Iterator pos = list.begin(); pos != fin; pos.next())
 if ((*pos).getName() == name)
 return pos;
 return fin;
}
```

```
int main() {
 List staff;
 staff.push_back(Employee("Michel",5000));
 staff.push_back(Employee("Roberta",5000));
 staff.push_back(Employee("Claudia",45000));
 staff.push_back(Employee("Mohamed",43000));
```

```
 Iterator claudia = find(staff, "Claudia");
 if (claudia != staff.end())
```

```
 ...
```

Retourne la position si trouvée.  
Sinon, retourne l'itérateur après la fin  
pour indiquer qu'on n'a pas trouvé

Vérifie si on a trouvé avant  
d'utiliser l'itérateur *claudia*.

# Exemple de parcours de liste avec boucle sur intervalle

- List et Iterator implémentent ce qu'il faut:
  - `for ( range_declararion : range_expr ) ...` équivalent à

```
auto&& __range = range_expr ; S'ils existent
auto __begin = __range.begin(); // = begin(__range);
auto __end = __range.end(); // = end(__range);
for (; __begin != __end; ++__begin) {
```

*range\_declaration* = \* \_\_begin;

```
} ...
```

Pas des vrais noms de variables  
(on ne peut pas obtenir l'itérateur)

- Donc on peut:

```
void afficher(List& list) {
 for (Employee& employee : list)
 cout << employee.getName() << " ";
 cout << endl;
}
```

# Classe List – implémentation

```
Iterator List::begin() {
 return Iterator(first_);
}
```

Retourne un itérateur pointant le premier élément (le constructeur d'*Iterator* prend en paramètre le pointeur position)

```
Iterator List::end() {
 return Iterator(Node::past_end);
}
```

Retourne un itérateur pointant **après** le dernier élément. On a une constante *past\_end* qui est ici simplement *nullptr*.

```
bool List::empty() const {
 return first_ == Node::past_end;
}
```

Si le premier est *past\_end*, la liste est vide. (Pourrait faire *begin()* == *end()* si on en avait des versions const.)

# Classe List – insertion d'un item à la fin

Bleu pour liens doubles

```
void List::push_back(const Employee& s) {
 Node* newnode = new Node(s);
 if (empty())
 first_ = newnode;
 else {
 last_->next_ = newnode;
 newnode->previous_ = last_;
 }
 last_ = newnode;
}
```

On crée un nouveau nœud.

Si la liste était vide, ce nœud est le premier.

Sinon, il est après l'ancien dernier nœud; l'ancien dernier est donc aussi avant ce nœud.

La liste doit maintenant pointer sur le « nouveau » dernier nœud

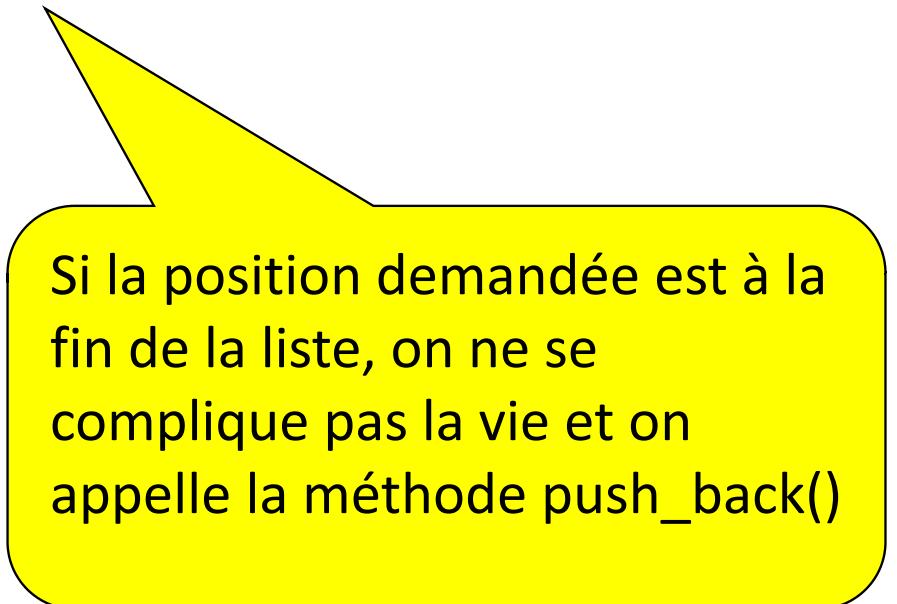
# Classe List – Insertion à la position

## indiquée par l'itérateur

Vert pour liens simples  
Bleu pour liens doubles

```
Iterator List::insert_after(Iterator iter, const Employee& s)
{
 if (iter.position_ == Node::past_end
 iter.position == last_) {
 push_back(s);
 return Iterator(last_);
 }
 ...
}
```

Retourne un itérateur pointant l'élément ajouté



# Insertion à la position indiquée par l'itérateur (suite)

...

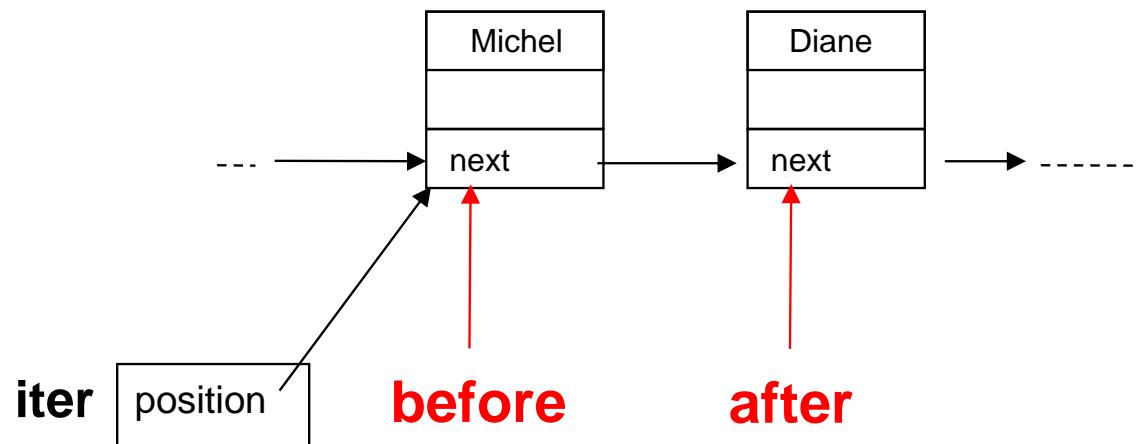
```
Node* before = iter.position_;
Node* after = before->next_;
```

Pointeurs vers les nœuds avant et après la position d'insertion

```
Node* newnode = new Node(s);
newnode->next_ = after;
before->next_ = newnode;
```

```
return Iterator(newnode);
```

}



# Insertion à la position indiquée par l'itérateur (suite)

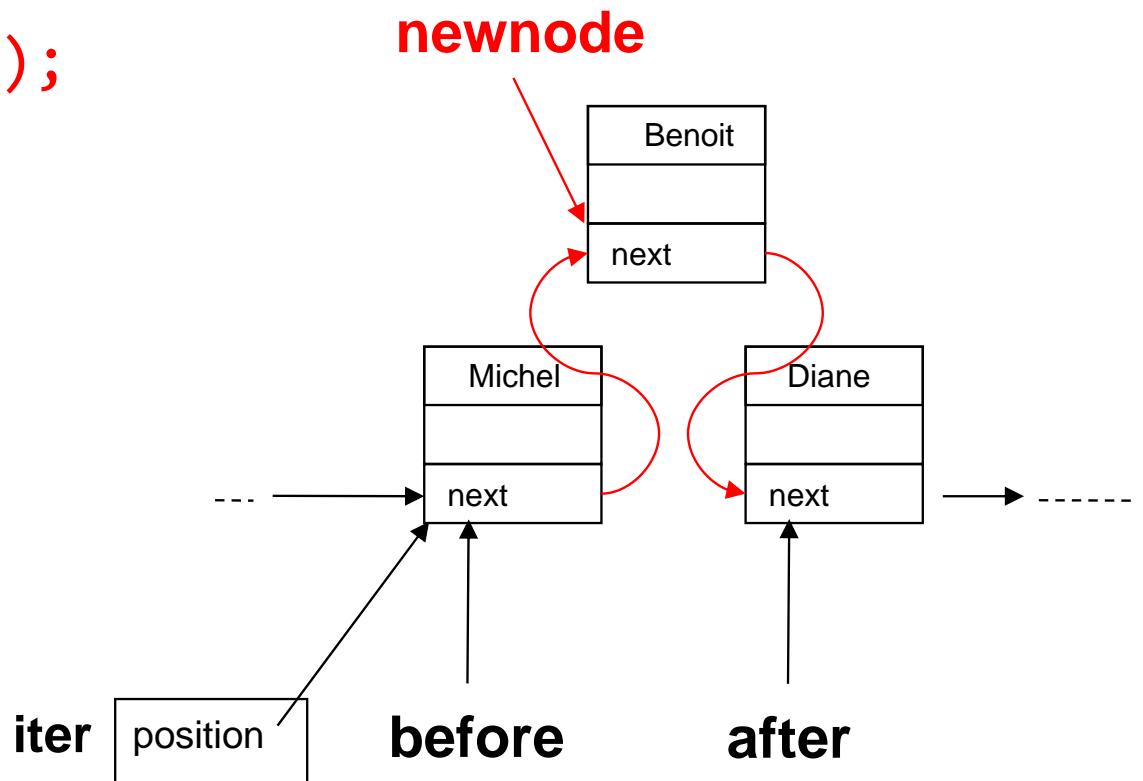
```

...
Node* before = iter.position_;
Node* after = before->next_;

Node* newnode = new Node(s);
newnode->next_ = after;
before->next_ = newnode;

return Iterator(newnode);
}

```



# Insertion à la position indiquée par l'itérateur (suite)

Liste liée DOUBLE

```

...
Node* after = iter.position_;
Node* before = after->previous_;
Node* newnode = new Node(s);
newnode->next_ = after;
newnode->previous_ = before;
after->previous_ = newnode;
if (before == Node::past_end)
 first_ = newnode;
else
 before->next_ = newnode;

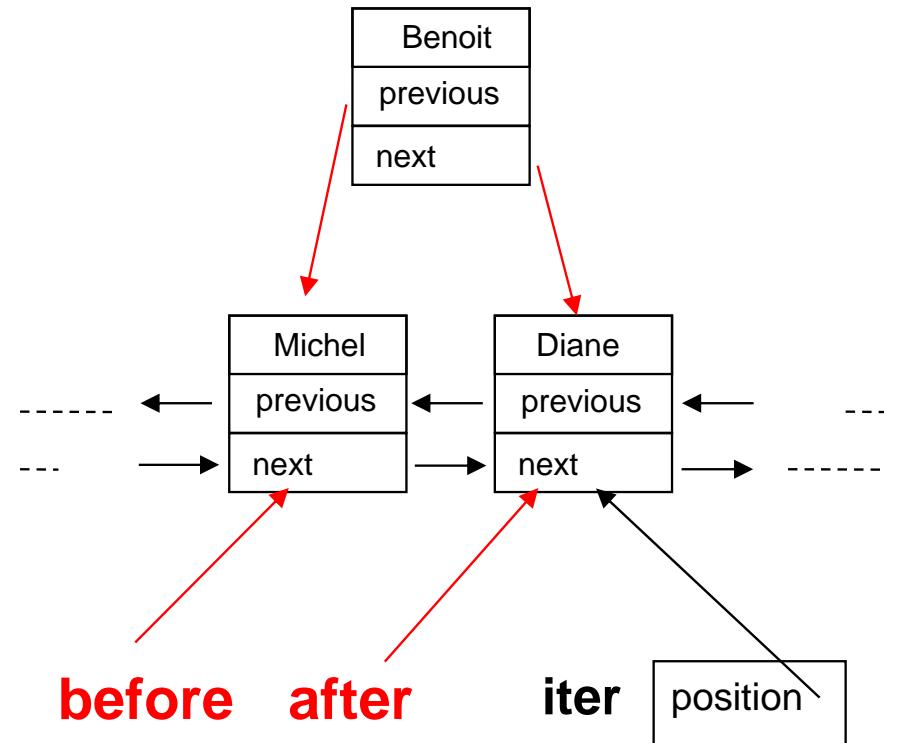
return Iterator(newnode);
}

```

On insère avant iter.

On crée le nouveau nœud et on le fait pointer vers les nœuds qui seront son précédent et son suivant

**newnode**



# Insertion à la position indiquée par l'itérateur (suite)

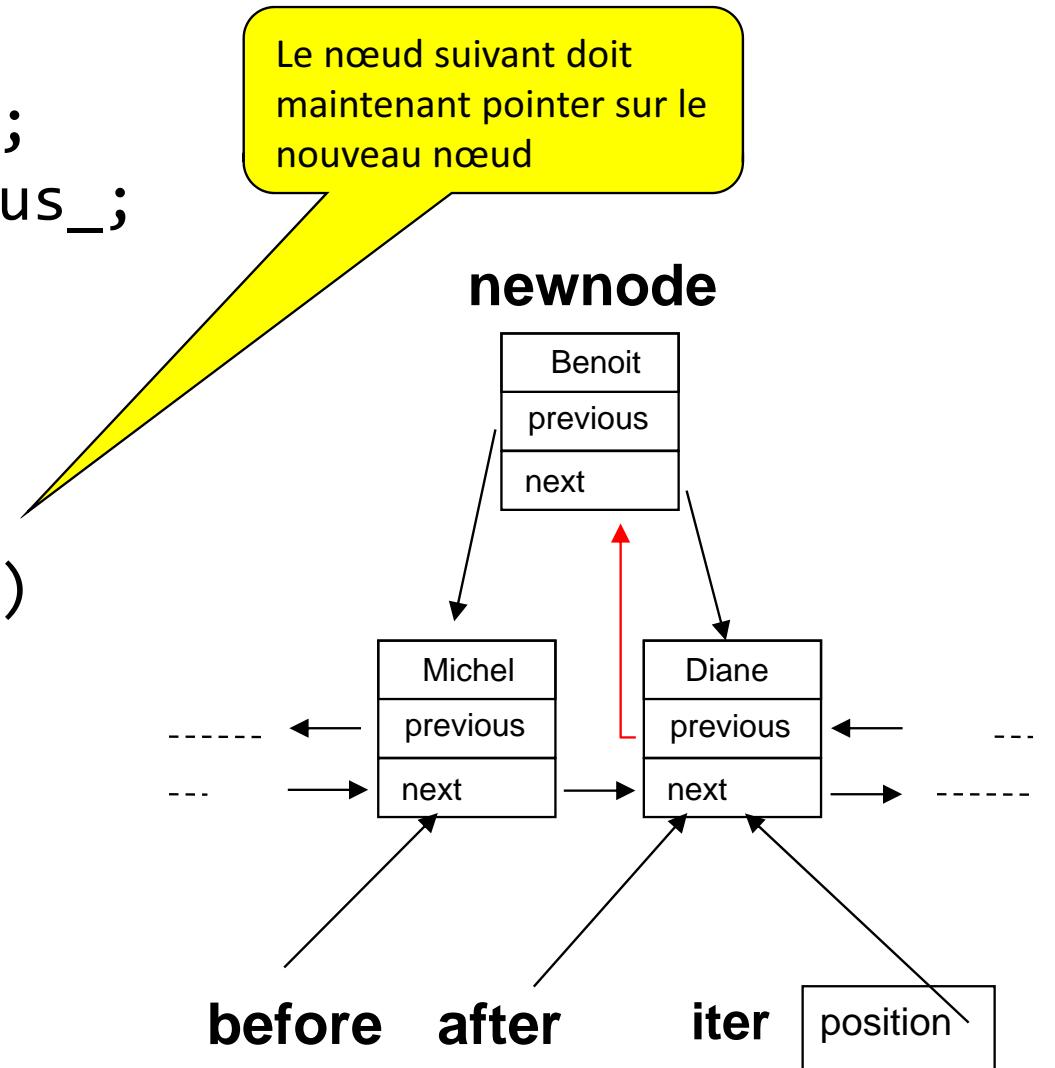
Liste liée DOUBLE

```

...
Node* after = iter.position_;
Node* before = after->previous_;
Node* newnode = new Node(s);
newnode->next_ = after;
newnode->previous_ = before;
after->previous_ = newnode;
if (before == Node::past_end)
 first_ = newnode;
else
 before->next_ = newnode;

return Iterator(newnode);
}

```



# Insertion à la position indiquée par l'itérateur (suite)

Liste liée DOUBLE

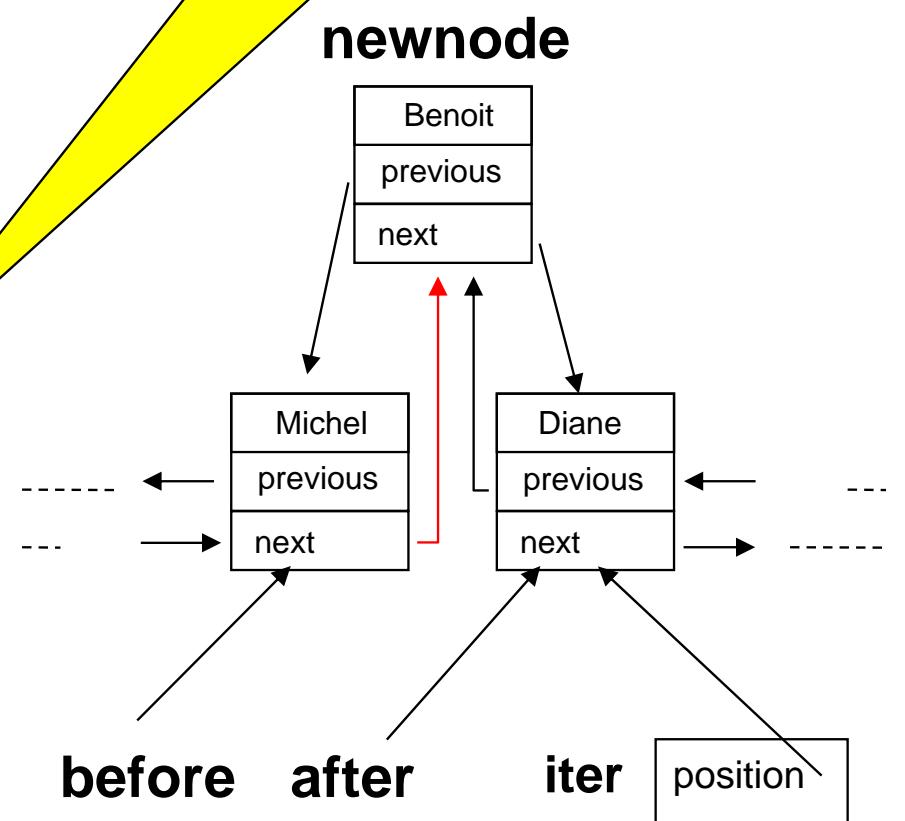
```

...
Node* after = iter.position_;
Node* before = after->previous_;
Node* newnode = new Node(s);
newnode->next_ = after;
newnode->previous_ = before;
after->previous_ = newnode;
if (before == Node::past_end)
 first_ = newnode;
else
 before->next_ = newnode;

return Iterator(newnode);
}

```

Le nœud précédent, s'il existe, doit lui aussi pointer sur le nouveau nœud



# Types membres:

## exprimer le lien entre List, Iterator, Node

- Chaque conteneur aura besoin de classes itérateur et détails de structure de données
- Classes membres pour lier ces classes:

```
template <typename T> Toute la classe est paramétrée, incluant les classes membres
class List2 {
private: Classe/struct privée
 struct Node { ... };
public: « iterator » pour suivre le standard C++ des conteneurs
 class iterator { https://en.cppreference.com/w/cpp/named_req/Container
 public: iterator(Node* position = Node::past_end);
 ...
 }; A accès à la classe privée car membre
};
int main() {
 List2<Employee>::iterator it;
À l'extérieur, doit spécifier le nom de classe dans laquelle la classe est
```

# Alias de types membres

Car templates avec paramètres différents, pour encapsulation, ou classe commune à plus d'un conteneur ...

- On peut vouloir séparer les déclarations des classes mais exprimer le même lien

Comme std::list dans MSVC et libstdc++

- Types membres:

```
template <typename T>
class List3 {
private:
```

Notre classe globale (nom avec \_impl pour indiquer privé, mais n'est pas vérifié par le compilateur); pourrait aussi utiliser un namespace, vu plus tard dans le cours.

```
 using Node = ListNode_impl<T>;
```

La classe n'est pas membre, il faut lui passer les paramètres dont elle a besoin

```
public:
```

```
 using iterator = ListIterator_impl<T>;
```

```
 ...
};
```

```
int main() {
 List3<Employee>::iterator it;
```

« iterator » est public, mais on garde la classe globale comme un détail qui ne devrait pas être utilisé de l'extérieur

## Bibliothèque de structures de données et algorithmes

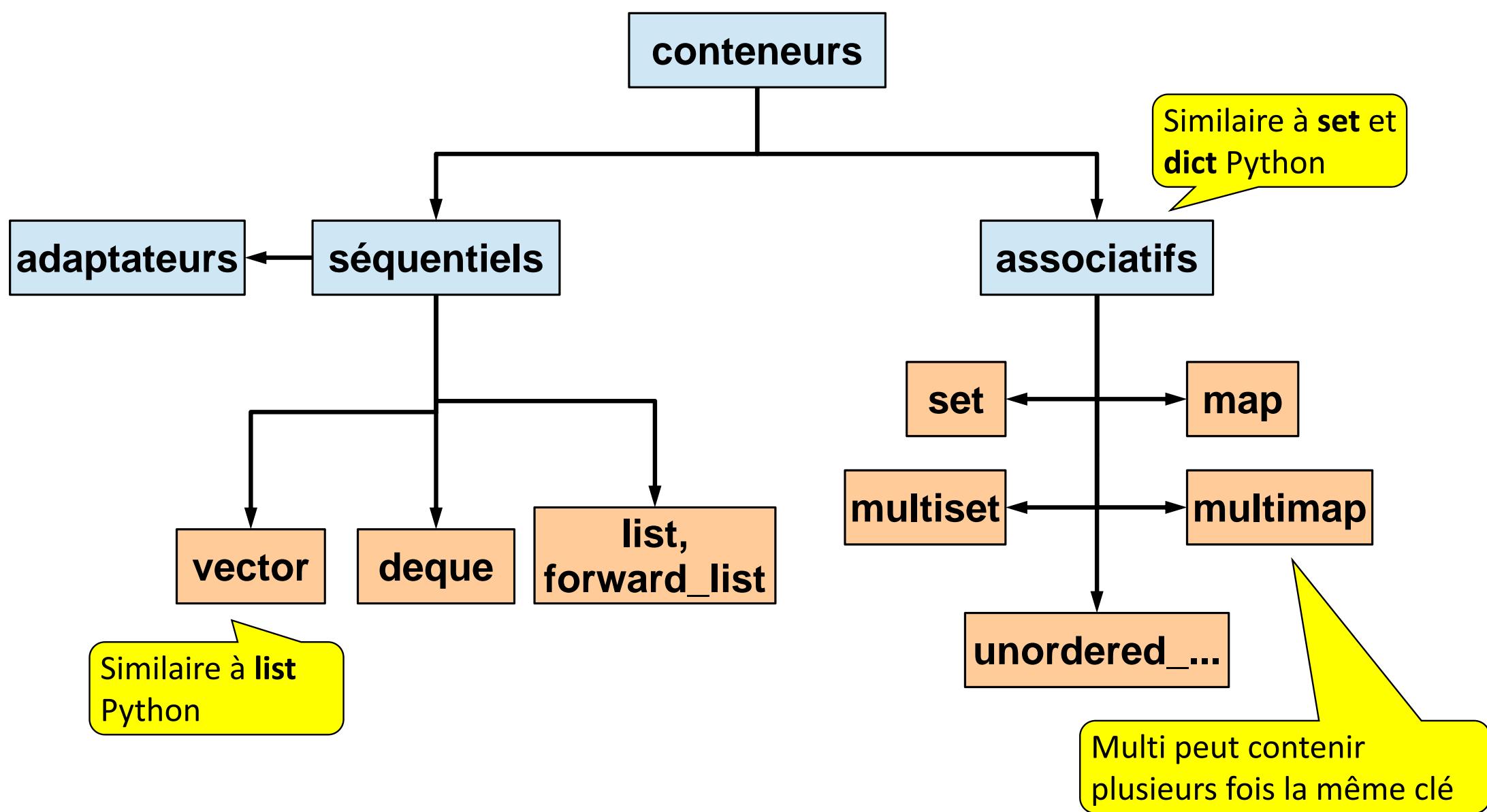
Conteneurs et leurs adaptateurs,  
concepts d'itérateurs, algorithmes,  
adaptateurs d'itérateurs

Générique

# C++ STL = Standard Template Library

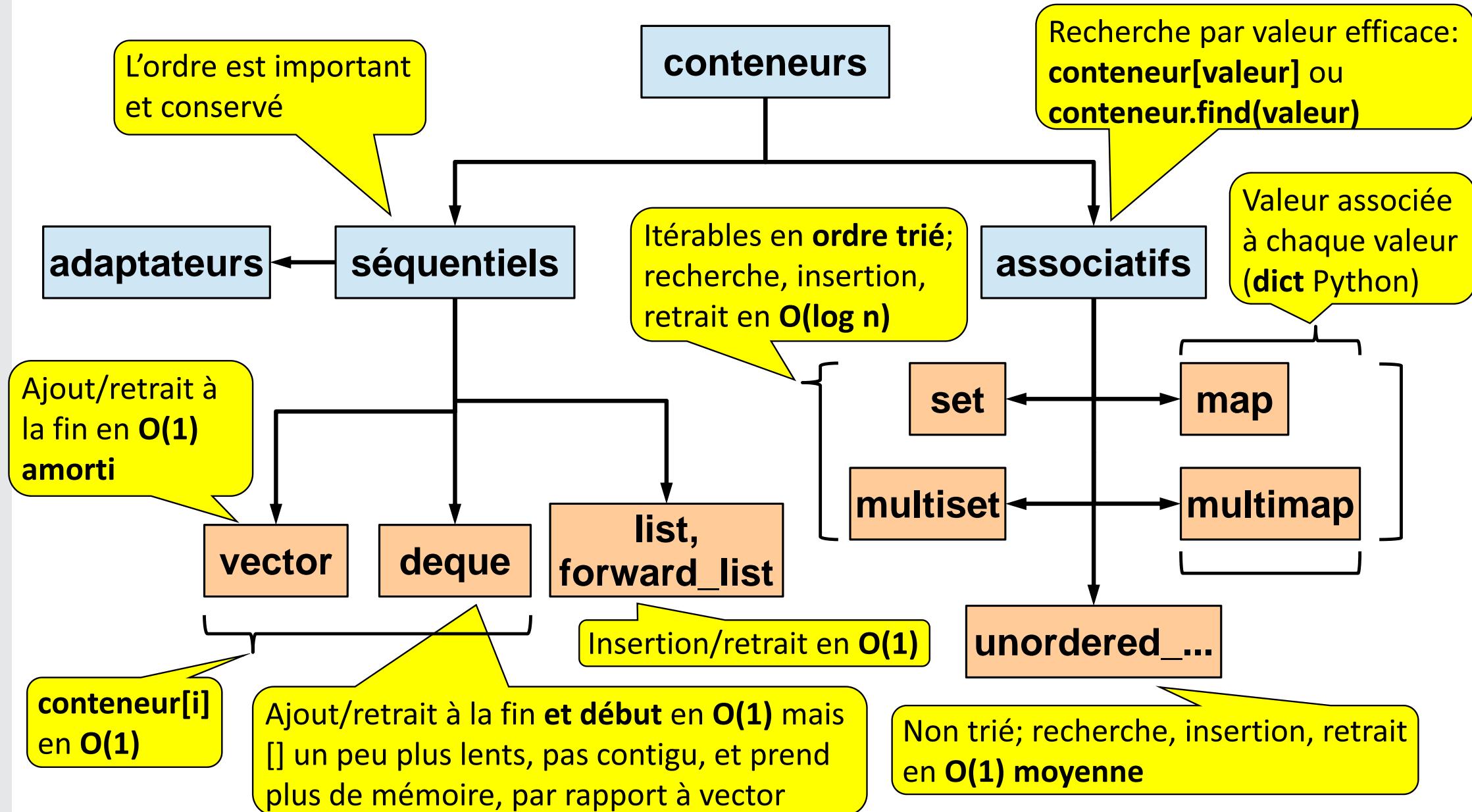
- Conteneurs (« containers ») et adaptateurs de conteneurs
  - Pour conserver des éléments
- Et leurs itérateurs (« iterators ») et adaptateurs d'itérateurs
  - Pour parcourir les conteneurs et faire certaines opérations simples (ex: insertion, retrait)
- Algorithmes
  - Pour faire des opérations sur les conteneurs (ex: recherche, tri)
- Concepts
  - Ce que les types doivent respecter

# Comment choisir le bon conteneur?

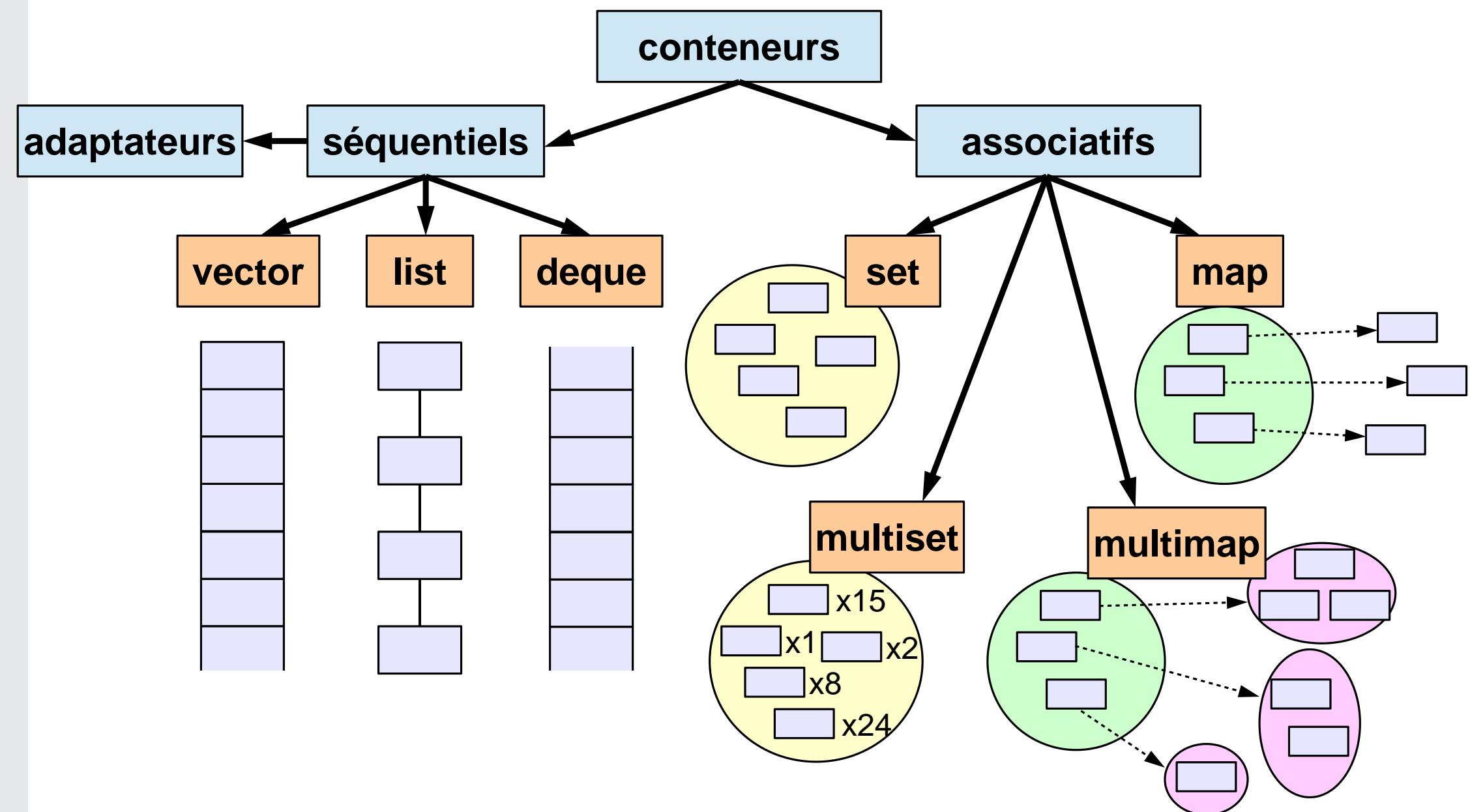


# Comment choisir le bon conteneur?

## À quoi ils sont bons?



# Représentation graphique des conteneurs



# Conteneurs séquentiels: quelques méthodes

Taille

- **size()**: combien d'éléments contenus
- **empty()**: vrai si le conteneur est vide

Toutes O(1) ou O(1) amorti,  
sauf insert / erase

Insertion

- **push\_back(x)**: ajouter **x** à la fin (vector, list, deque)
- **push\_front(x)**: ajouter **x** au début (list, deque)
- **insert(it, x)**: ajouter **x** avant l'élément pointé par **it**

Retrait

- **pop\_back()**: retirer à la fin (vector, list, deque)
- **pop\_front()**: retirer au début (list, deque)
- **erase(it)**: retirer l'élément pointé par **it**
- **clear()**: vider le conteneur

insert / erase sont O(1)  
sur liste, O(n) sur vector  
et deque

Accès

- **back()**: valeur de l'élément à la fin (vector, list, deque)
- **front()**: valeur de l'élément au début (list, deque)

- Plus? <http://fr.cppreference.com/w/cpp/container>

# Pourquoi `pop_...` ne retourne pas la valeur de l'élément?

- Pour performance
  - L'objet est détruit dans le conteneur → ne peut pas le retourner par référence
  - Un retour par valeur prend du temps pour la copie
  - On n'utilise pas toujours la valeur retournée
- On a donc une autre méthode pour la valeur
  - Exemple pour afficher et enlever les éléments:

```
while (!conteneur.empty()) {
 cout << conteneur.back() << endl; // ou .front()
 conteneur.pop_back(); // ou .pop_front()
}
```

# Adaptateurs de conteneurs (séquentiels)

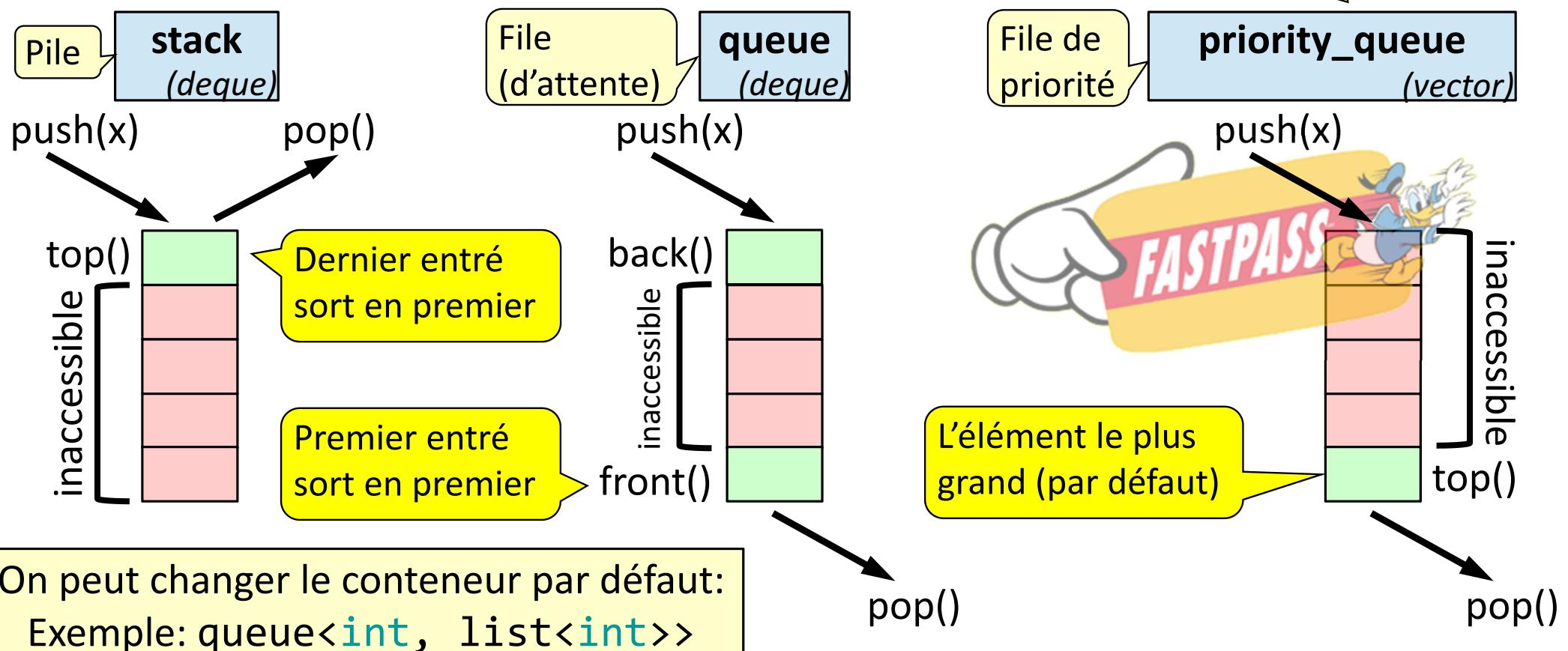
- Adapte pour une utilisation plus spécifique

- Seules les méthodes pour cette utilisation sont accessibles

N'ont pas d'itérateurs

Méthodes en O(1) par défaut, sauf:

Push / pop O(log n)



# Conteneurs associatifs

- Généralisation de l'indexation
  - Les conteneurs séquentiels sont indexés par un entier ( $0, 1, 2, \dots$ ), les conteneurs associatifs le sont par n'importe quel type (type de l'élément choisi pour clé) qui supporte `<` ou `==` et `hash` == et « hash » pour unordered...  
Par défaut, peut changer la comparaison
- La clé est toujours **const** Changer sa valeur doit déplacer l'élément, on le fait en déplaçant l'élément explicitement
- La clé détermine la position d'insertion ou de retrait (vs. l'index pour les séquentiels)

# Conteneurs associatifs

## map (et multimap)

Chap.5

- Contient des **pair**, dont la **clé est constante**

```
map<string,double> uneMap = {{"x",1.25}, {"hello",1.0}};
```

Construction avec initialisation

```
uneMap["bonjour"] = 2.5;
```

Met une valeur dans le conteneur

Itérateur sur le premier élément

```
map<string,double>::iterator it = uneMap.begin();
```

```
auto& [cle,valeur] = *it;
```

Prend cle et valeur par référence

```
valeur = 3.25;
```

On peut changer la valeur

```
cout << uneMap["bonjour"] << endl;
```

Et voir que la valeur a changée

```
// cle = "allo";
```

```
// *it = pair{"allo", 4.75};
```

Erreur de compilation, ne peut pas changer la clé

```
uneMap["allo"] = valeur;
```

On peut ajouter et enlever l'ancienne

```
uneMap.erase(it);
```

```
it = uneMap.begin();
```

**erase** a invalidé l'itérateur de l'élément enlevé

```
map<string,double>::node_type noeud = uneMap.extract(it);
```

```
noeud.key() = "hi";
```

On peut extraire le nœud, le modifier, et le réinsérer avec move (C++17)

```
uneMap.insert(move(noeud));
```

```
for (auto&& [k,v] : uneMap) cout << k << ":" << v << " ";
```

Itère sur les paires en ordre trié

# Conteneurs associatifs

## set (et multiset)

- Similaire à **map** mais la clé et valeur sont la même
  - La clé/valeur est aussi **const**
- Ne surcharge pas []
  - `.find(valeur)` → un itérateur
  - `.count(valeur)` → combien de fois la valeur se trouve
  - `.contains(valeur)` → vrai/faux
  - `.insert(valeur)` ajoute
  - `.erase(it)` enlève
  - `.extract ...` similaire à **map**

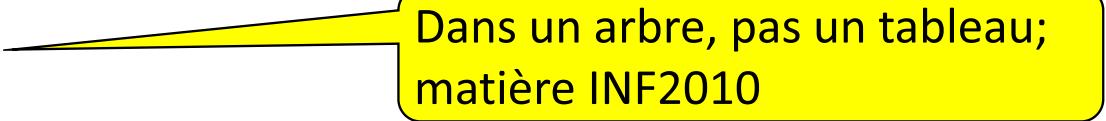
0 ou 1 pour **set**,  
**multiset** permet > 1

C++20

C++17

# Conteneurs associatifs

## pas unordered \_ ...

- Les map, multimap, set, multiset comparent les éléments pour les trier 

Dans un arbre, pas un tableau;  
matière INF2010
- Par défaut la surcharge d'operator< est suffisante
  - Il vérifie l'« équivalence » == comme !(a < b) && !(b < a)
- Si je veux un autre ordre?
  - map<string,double> est équivalent à  
map<string,double,less<string>>

Un type d'objet fonction, aussi dit foncteur

L'appel de l'objet compare les valeurs

Construction d'un objet

Comment est défini ce type?

# Conteneurs associatifs

## pas `unordered_...` Type pour comparaison

- Exemple de foncteur de comparaison

Pas besoin d'encapsulation, juste un opérateur public

```
struct MaComparaison {
 bool operator() (const string& a, const string& b) const
 {
 return a > b;
 }
};

int main()
{
 map<string,double,MaComparaison> uneMap = {
 {"x",1.25}, {"hello",1.0}
 };
 map<string,double,MaComparaison>::iterator it = uneMap.begin();
 ...
}
```

Constructible par défaut

Surcharge l'opérateur d'appel de fonction

Ce qu'on veut pour comparer

Utilise la comparaison ci-dessus

Tous les types membres pourraient changer et deviennent longs, `auto` est pratique

# Conteneurs associatifs

## unordered\_...

- Les unordered\_map, unordered\_multimap, unordered\_set, unordered\_multiset ne trient pas mais organise les éléments en seaux numérotés
- Une fonction de hash sert à déterminer le numéro de seau
  - Fonction de clé qui retourne un entier (size\_t)
  - Doit donner le même entier pour des clés considérées équivalentes
  - Doit avoir une faible probabilité de donner le même entier pour des clés non équivalentes
- == pour comparer à l'intérieur du seau

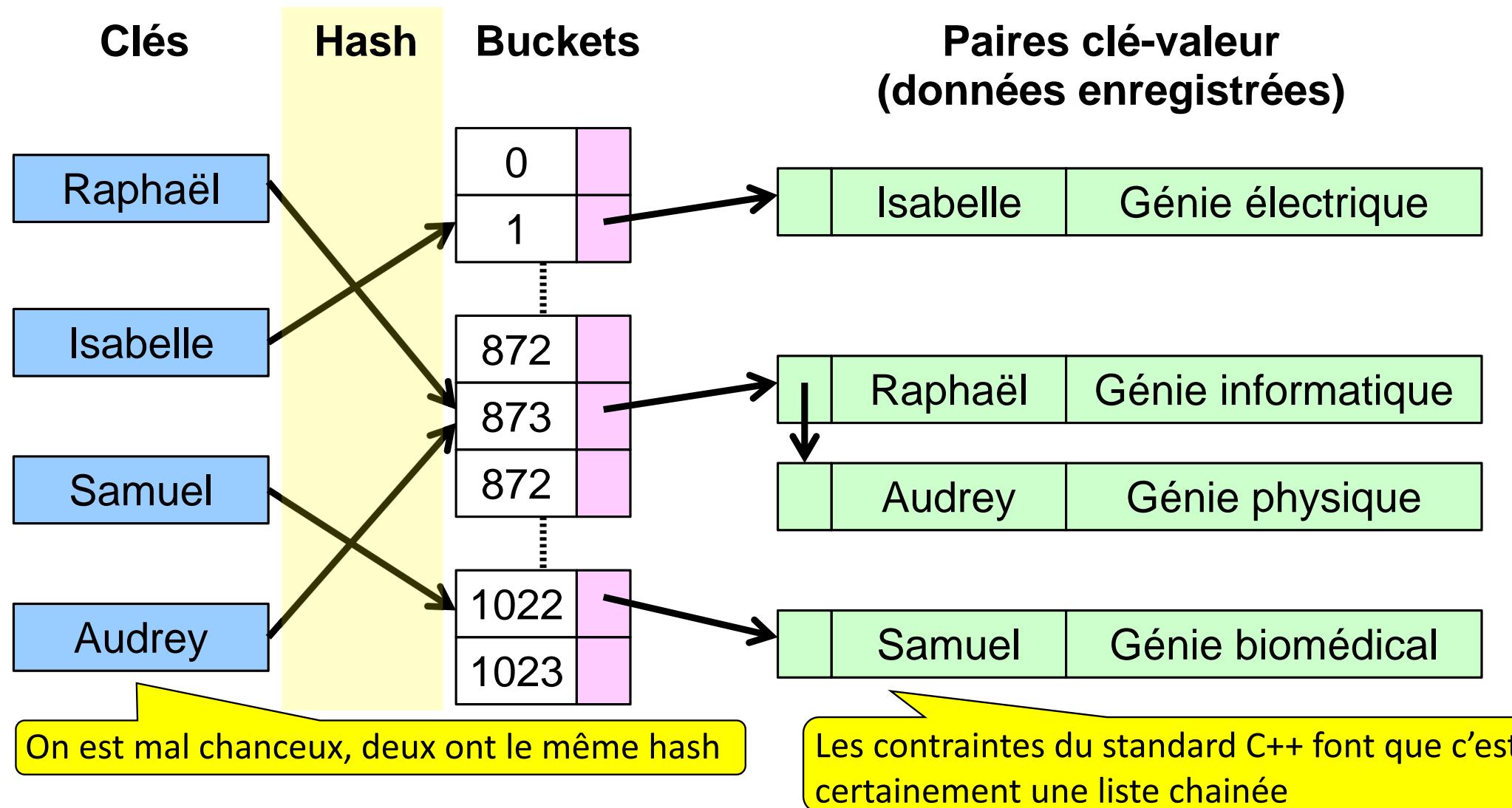
« buckets »

On espère avoir chaque élément dans un seau différent

# Conteneurs associatifs

## unordered\_... hachage

- Représentation du hachage d'une map:



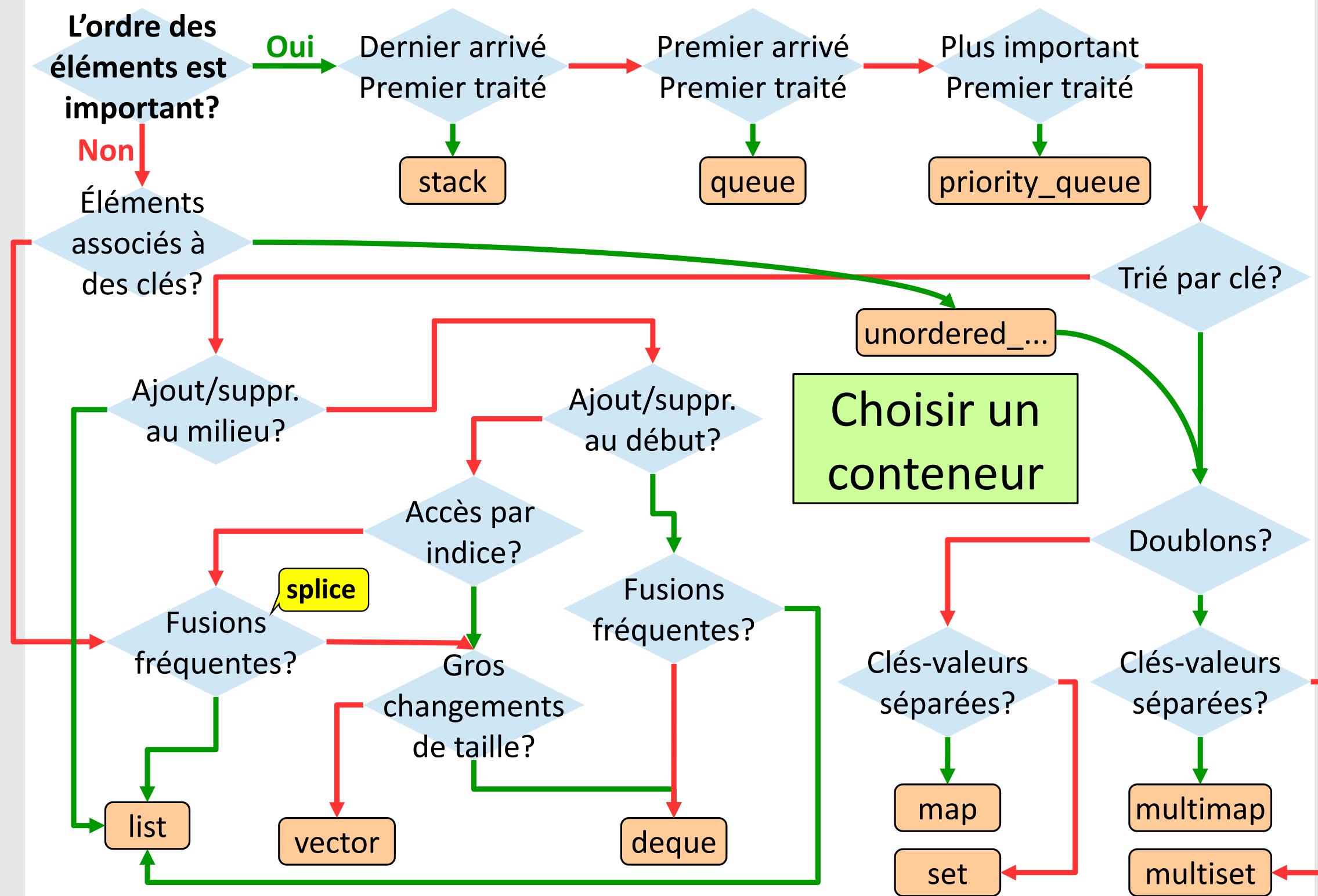
# Conteneurs associatifs

## unordered\_... pour notre propre type

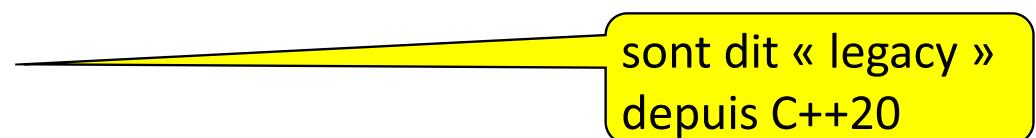
```
// Basé sur https://en.cppreference.com/w/cpp/utility/hash
struct Nom {
 string prenom, nom;
 bool operator==(const Nom&) const = default; Doit supporter == (manière C++20)
};

namespace std { Ici on spécialise directement hash<Clé> dans std:: pour notre type
 template<> struct hash<Nom> { Foncteur
 size_t operator()(const Nom& nom) const noexcept
 {
 return hash<string>{}(nom.prenom)
 ^ (hash<string>{}(nom.nom) << 1);
 }
 };
}

int main() {
 unordered_map<Nom, int> matricules = { {"John", "Doe"}, 12345 };
 cout << matricules[{"John", "Doe"}] << endl; Affiche quoi?
}
```



# Les type d'itérateurs

- Les itérateurs des conteneurs de la STL sont classés en catégories
  - LegacyInputIterator
  - LegacyOutputIterator
  - LegacyForwardIterator
  - LegacyBidirectionalIterator
  - LegacyRandomAccessIterator

sont dit « legacy » depuis C++20
- C++20 a des concepts d'itérateursDans `<iterator>`
  - Plusieurs sont « équivalents », mais maintenant définis en programme (au lieu d'en mots) et vérifiés
    - ex: `input_iterator`, `output_iterator`, `forward_iterator`, ...

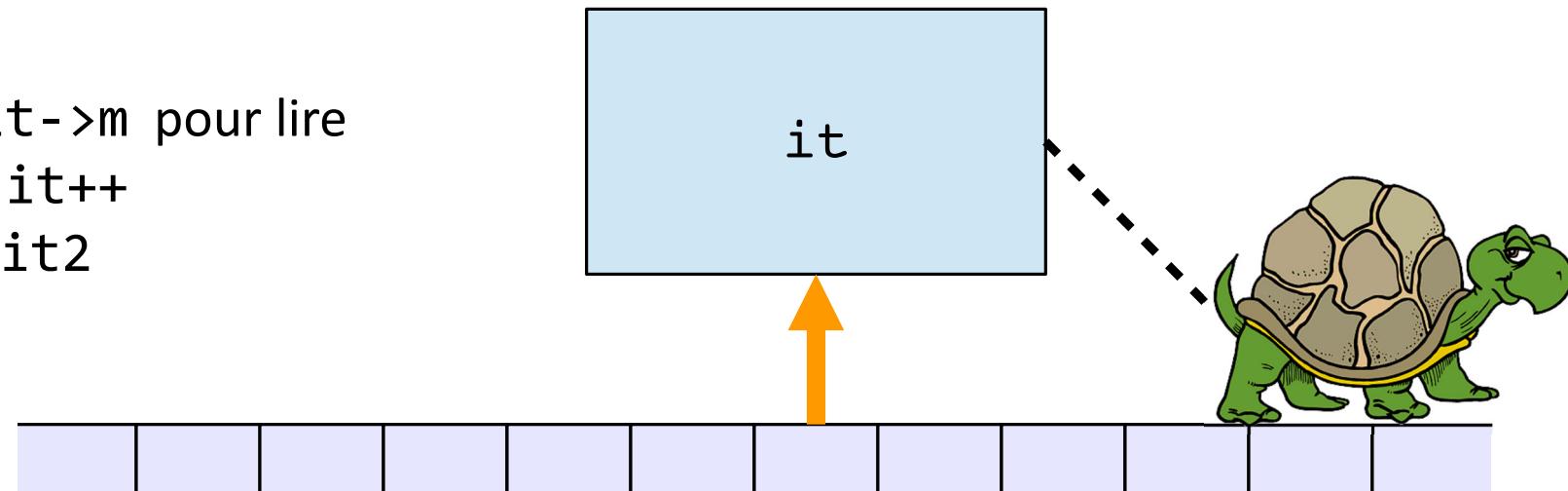
# Les type d'itérateurs:

## Itérateur d'entrée (input iterator)

- Donne accès à une source de données (lecture!)
- Cette source peut être un conteneur, un stream, etc.
- Permet de lire au moins **une fois** du début à la fin (pas nécessairement de revenir à une ancienne position)

Permet:

\*it et it->m pour lire  
++it et it++  
it1 != it2



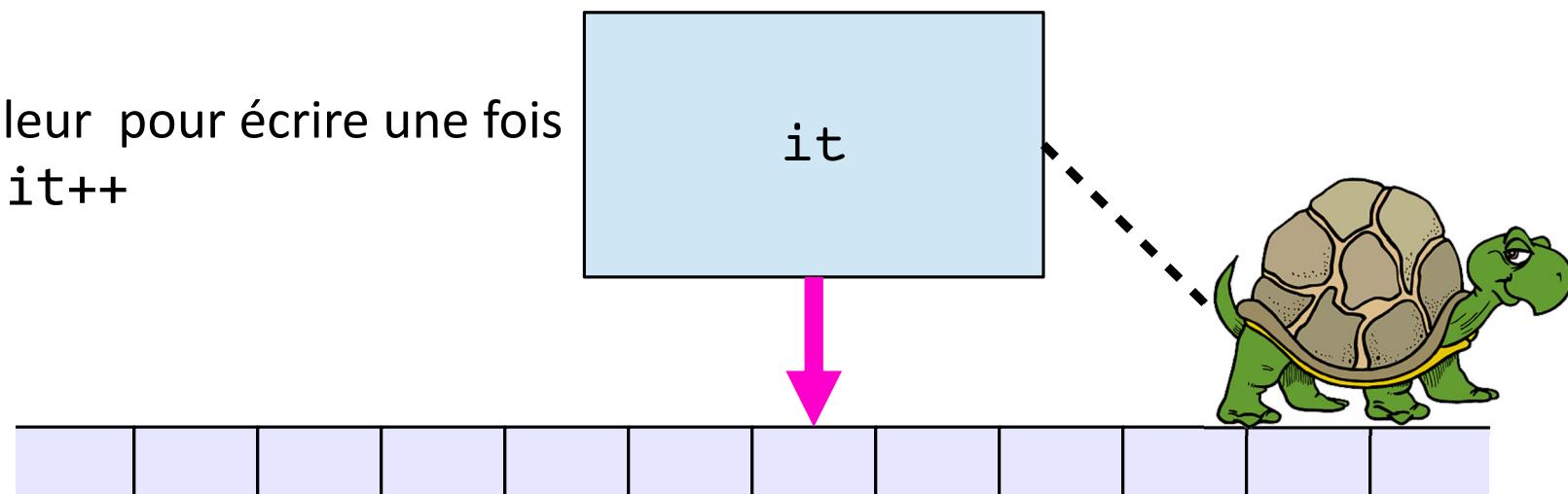
# Les type d'itérateurs:

## Itérateur de sortie (output iterator)

- Donne accès à des collecteurs de données (emplacement pour stocker des informations) (écriture!)
- Le collecteur peut être un conteneur, un stream, etc.
- Comme pour l'itérateur d'entrée, on peut passer **une fois** du début (pas nécessairement réécrire)

Permet:

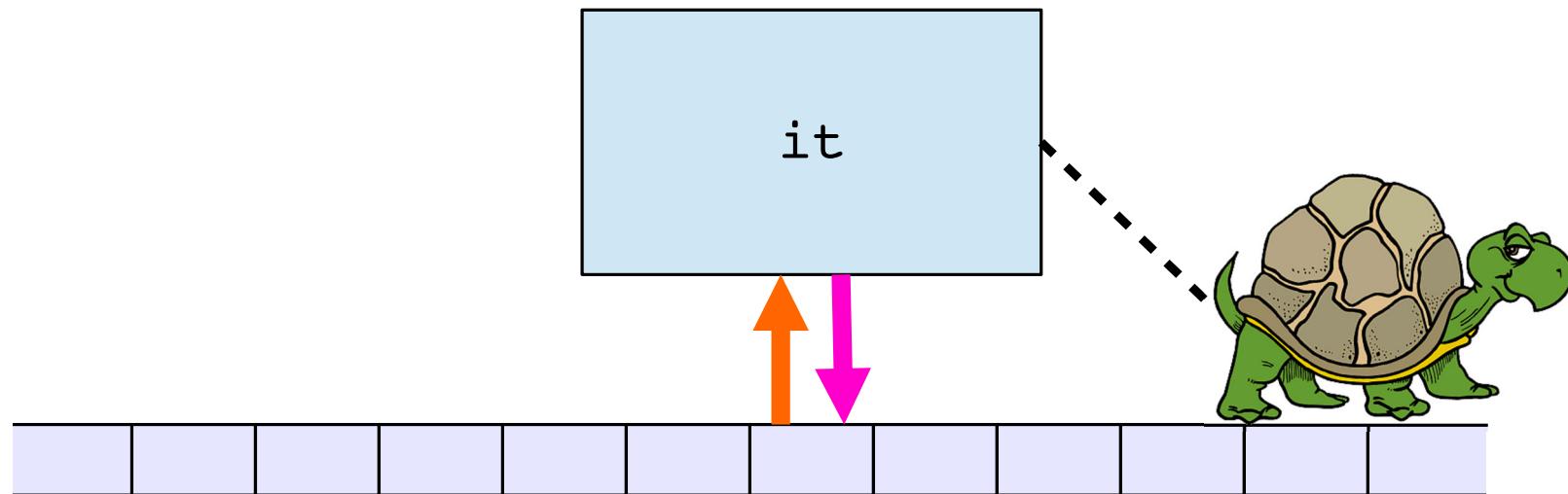
`*it` = valeur pour écrire une fois  
`++it` et `it++`



# Les type d'itérateurs:

## Itérateur avançant (forward iterator)

- Permettent tout de itérateurs d'**entrée** et de **sortie**
- Peuvent traverser un intervalle **plusieurs fois** (si on repart d'un même itérateur d'origine) via incrémentation, pour écrire/lire plusieurs fois



# Les type d'itérateurs:

## Itérateur bidirectionnel (bidirectional ...)

- Comme itérateur avançant mais peut en plus reculer --

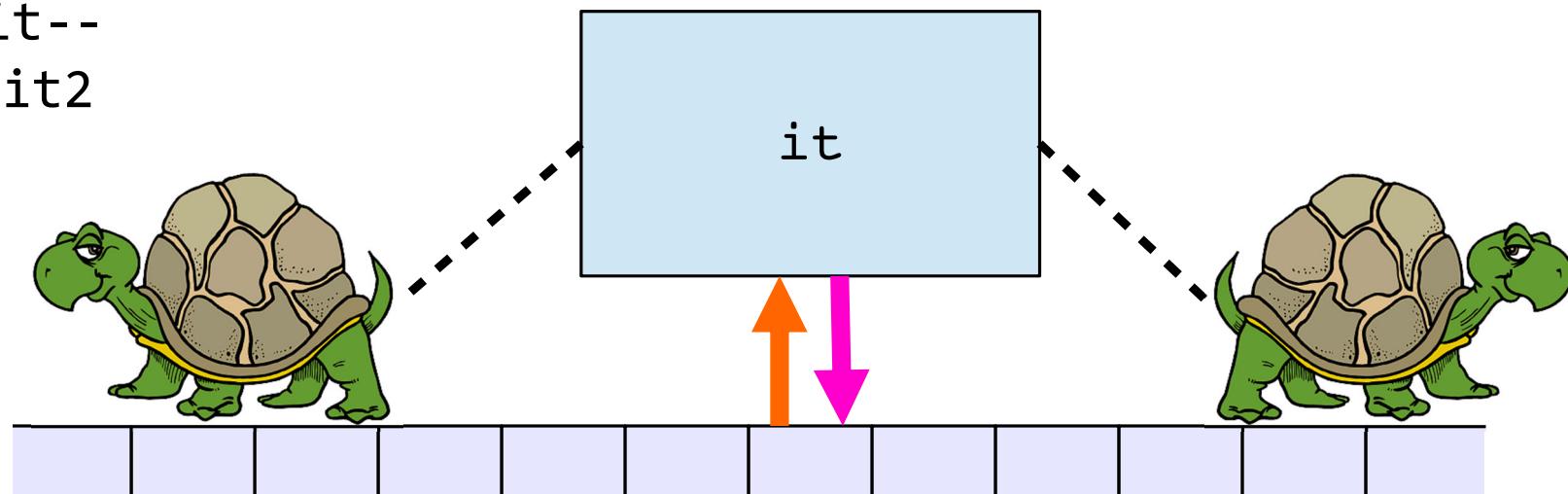
Permet:

\*it et  $it \rightarrow m$  pour lire et écrire

$++it$  et  $it++$

$--it$  et  $it--$

$it1 \neq it2$



# Les type d'itérateurs: Itérateur à accès aléatoire (random access ...)

- Comme bidirectionnels, plus...
  - Peuvent sauter de n'importe quelle position à n'importe quelle autre en  $O(1)$
  - Peuvent être comparés  $<$ ,  $\leq$ ,  $>$ ,  $\geq$

Permet en plus:

`it[n]; it += n; it + n; n + it;`

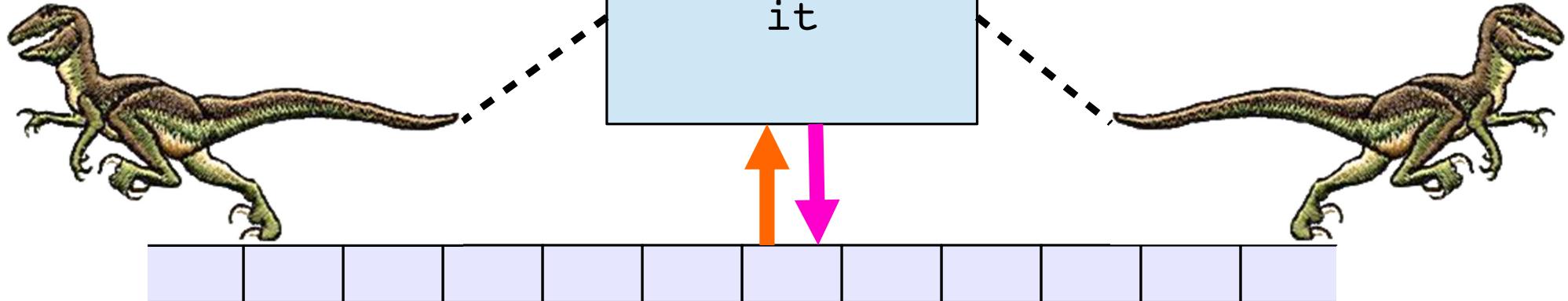
`it -= n; it - n;`

`it2 - it1`

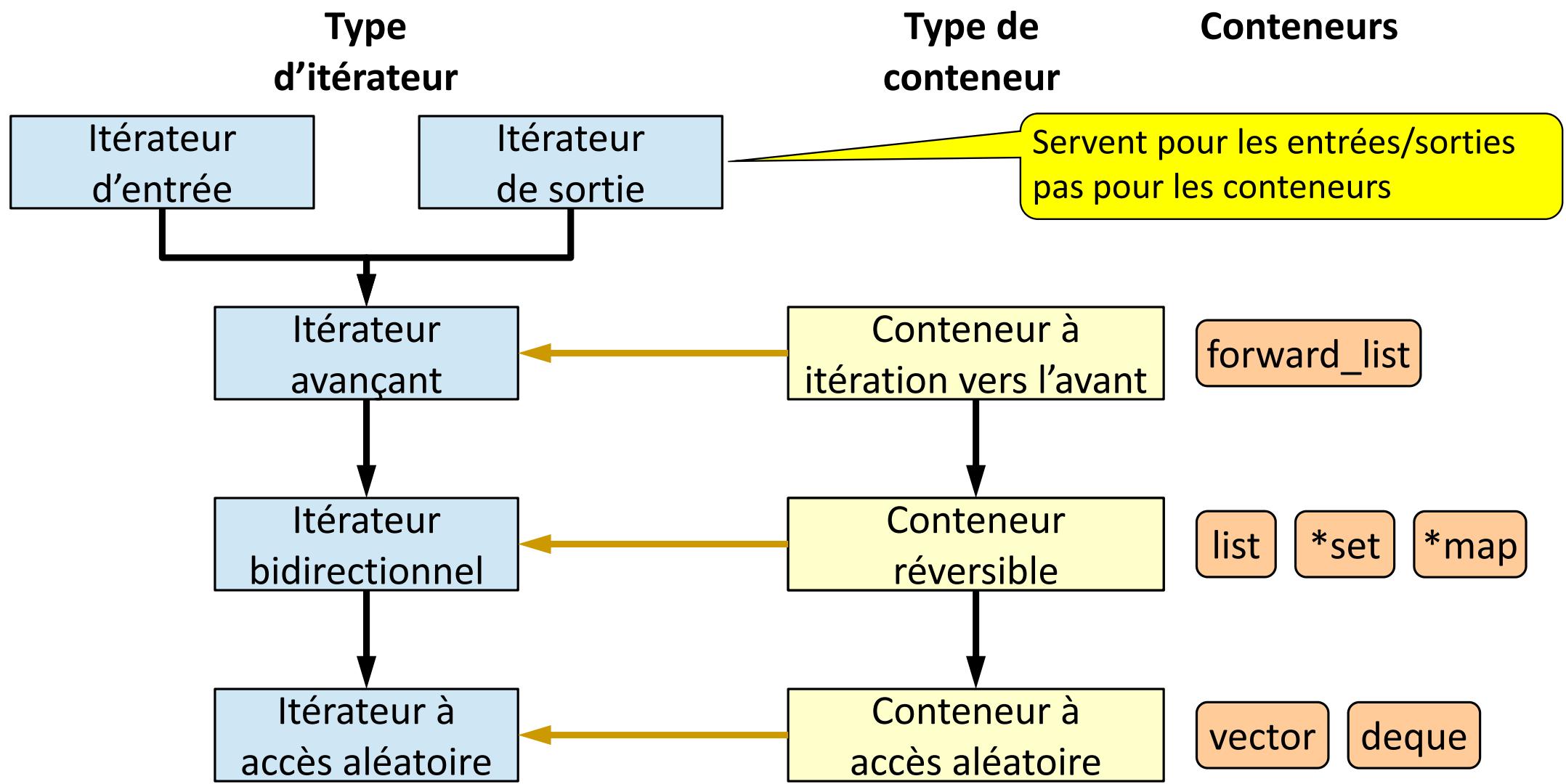
`it1 <=> it2`

Nombre d'éléments entre les deux

Un pointeur brut est un itérateur à accès aléatoire sur un tableau contigu



# Quel conteneur fournit quel type d'itérateur?



# Invalidation d'itérateurs sur ajout/retrait d'éléments au conteneur

- Retrait invalide toujours les itérateurs et références à ce qui est enlevé
- list, forward\_list, set, map : rien de plus est invalidé
- unordered\_set, unordered\_map
  - Ajout invalide tout itérateur si rehash; références restent valides
- Vector
  - Ajout/retrait: itérateurs/références après le point d'ajout/retrait sont invalidés (incluant **end**)
  - Si réallocation sur ajout, tout est invalidé
- Deque: tout est invalidé sauf si au début ou fin
  - Ajout au début/fin n'invalide pas les références
  - Retrait au début/fin invalide seulement pour les éléments enlevés (et **end** si à la fin)

Réorganisation des seaux  
quand trop pleins

# Invalidation d'itérateurs sur ajout/retrait

## d'éléments au conteneur, attention

```
vector<string> v = {"bonjour", "allo", "salut"};
```

```
auto fin = v.end();
for (auto it = v.begin(); it != fin; ++it)
 v.push_back(*it + " toi");
```

Prend **end** avant la boucle pour boucler sur les éléments avant ajouts

```
for (auto it = v.begin(); it != v.end(); ++it)
 if ((*it)[0] < 'c')
 v.erase(it);
```

Invalide **it** et **fin** : undefined behavior

Invalide **it** : undefined behavior

```
v.reserve(v.size() * 2);
auto fin = v.end();
for (auto it = v.begin(); it != fin; ++it)
 v.push_back(*it + " toi");
```

Augmente capacité assez pour les push\_back dans la boucle

```
for (auto it = v.begin(); it != v.end();)
 if ((*it)[0] < 'c')
 it = v.erase(it);
else
 ++it;
```

Ok car capacité suffisante

Pas le ++ ici

Cette boucle est correcte mais possiblement lente si v long.  
Elle est O de quoi?

Itérateur sur le prochain élément

# Les algorithmes

- Les algorithmes de la STL sont des procédures qui sont appliquées sur les conteneurs pour traiter leurs données
- Les algorithmes utilisent des itérateurs pour travailler sur les conteneurs, souvent comme:

| Algorithme                                   | Début de l'intervalle | Fin de l'intervalle | Fonction/foncteur |
|----------------------------------------------|-----------------------|---------------------|-------------------|
| for_each(v.begin(), v.end(), negate<int>()); |                       |                     |                   |

- Les algorithmes ont généralement une nouvelle version en C++20 dans std::ranges :

```
ranges::for_each(v, fonction, fonctionProjection);
```

Répond à un concept de range

Pas encore supporté complètement

Optionnelle

# Les algorithmes

## tri, min, max

```

template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);

template <class ForwardIt, class Compare>
bool is_sorted(ForwardIt first, ForwardIt last, Compare comp);

template <class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last);

template <class ForwardIt>
ForwardIt max_element(ForwardIt first, ForwardIt last);

int main() {
 int tab[] = { 5, 1, 3, 8, 4, 12, 8, 15, 8 };
 int* itMin = min_element(tab, tab + size(tab));
 int* itMax = max_element(tab, tab + size(tab));
 assert(*itMin == 1 && *itMax == 15);
 sort(tab, tab + size(tab)); // {1, 3, 4, 5, 8, 8, 12, 15}
 assert(is_sorted(tab, tab + size(tab)));
}

```

Dans <algorithm>

Ne fonctionne pas sur `list`, pourquoi?

On utilise des pointeurs comme itérateurs

# Les algorithmes

## tri, min, max; même exemple en C++20

```
int main() {
 auto tab = to_array({ 5, 1, 3, 8, 4, 12, 8, 15, 8 });
 auto itMin = ranges::min_element(tab);
 auto itMax = ranges::max_element(tab);
 assert(*itMin == 1 && *itMax == 15);
 ranges::sort(tab); // {1, 3, 4, 5, 8, 8, 12, 15}
 assert(ranges::is_sorted(tab));
}
```

Prend un ranges::random\_access\_range  
(un concept)

Pas besoin de donner deux itérateurs,  
on donne quelque chose qui supporte  
**begin** et **end** (un **range**), tel que tout  
conteneur, **span** et autres.

# Variantes des types d'itérateurs

- Sauf **sort**, les opérations précédentes devraient pouvoir se faire sur un conteneur **const**
- En plus du type **iterator**, les conteneurs donnent accès à d'autres types d'itérateurs:
  - **const\_iterator**
    - Permet de respecter l'engagement **const**
  - **reverse\_iterator**
    - Permettent de parcourir un conteneur à l'envers avec `++`
    - Utilise **rbegin()** et **rend()** pour récupérer l'intervalle d'un conteneur (qui doit être bidirectionnel au moins)
  - **const\_reverse\_iterator**
    - Cumule **const\_iterator** et **reverse\_iterator**

# Variantes des types d'itérateurs, exemple

```
int main() {
 auto tab = to_array({ 5, 6, 7, 8 });
 for (auto it = tab.crbegin(); it != tab.crend(); ++it)
 cout << *it << " ";
 cout << endl;
}

for (int v : ranges::reverse_view(tab))
 cout << v << " ";
cout << endl;
```

const reverse begin part de la fin

Affiche quoi?

++ recule dans le conteneur

C++20

# Les algorithmes

## recherche, compare, copie, remplir

count(itDébut, itFin, val) Compte combien sont == val ou prédicat est vrai  
 count\_if(itDébut, itFin, prédicatUnaire) Fonction valeur → bool  
 find(itDébut, itFin, val) Retourne un itérateur sur le premier trouvé  
 find\_if(itDébut, itFin, prédicatUnaire)  
 equal(itDébut, itFin, itDébutAutre) Vrai si les éléments de deux intervalles sont égaux  
 copy(itDébut, itFin, itDestination)  
 copy\_n(itDébut, nÉléments, itDestination) Copie, ou copie si vrai  
 copy\_if(itDébut, itFin, itDestination, prédicatUnaire)  
 fill(itDébut, itFin, val) ou fill\_n Remplit de valeurs  
 generate(itDébut, itFin, générateur) of generate\_n  
Fonction sans paramètre → valeur

# Les algorithmes

## attention, ils n'insèrent pas

[https://en.cppreference.com/w/cpp/iterator#Iterator\\_adaptors](https://en.cppreference.com/w/cpp/iterator#Iterator_adaptors)

- On doit utiliser un adaptateur d'itérateur

```
int main() {
 vector v1 = { 5, 6, 7, 8, 9, 10 };
 vector<int> v2;
 // copy_n(v1.begin(), 3, v2.begin());
 copy_n(v1.begin(), 3, back_inserter(v2));
 copy_if(v1.begin(), v1.end(), back_inserter(v2),
 [] (int x) { return x%2 == 0; });
 cout << "Entrer deux entiers: ";
 copy_n(istream_iterator<int>(cin), 2, back_inserter(v2));
 generate_n(back_inserter(v2), 4,
 [x=16]() mutable { return x *= 2; });
 copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
 cout << endl;
}
```

Plante car tente d'écrire dans des cases de v2 qui n'existent pas

back\_inserter utilise push\_back

Notre prédicat unaire

istream\_iterator lit avec >>

Notre générateur

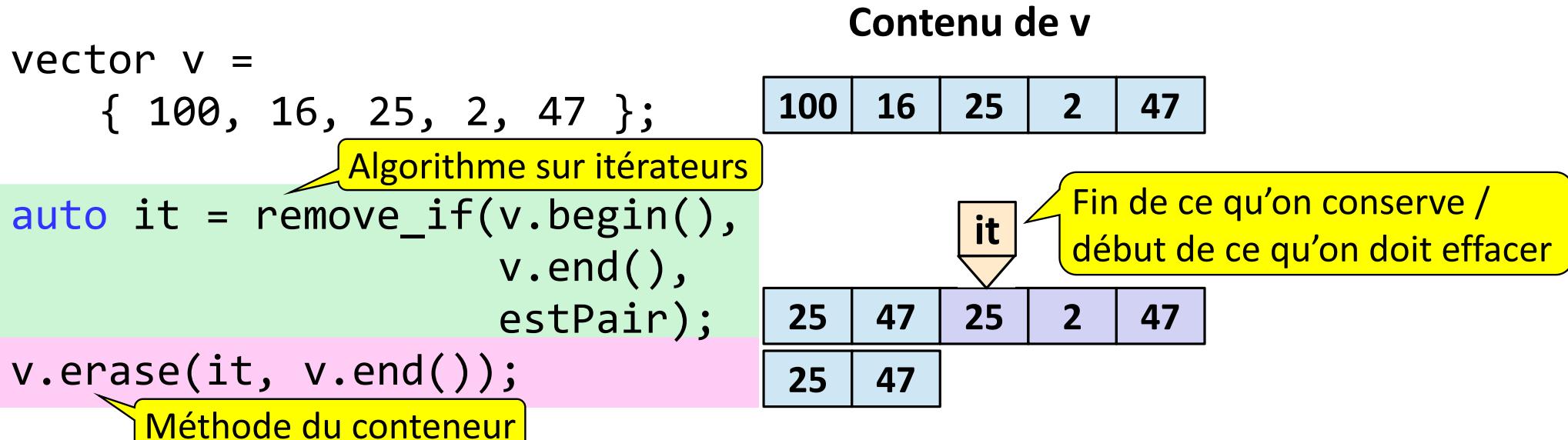
ostream\_iterator écrit avec << et texte const char[] après chaque élément

# Les algorithmes

## attention, ils ne retirent pas

- Les itérateurs ne savent pas comment effacer, et on ne donne que des itérateurs à l'algorithme
- « **erase–remove idiom** »: **remove** met les éléments conservés au début et retourne un itérateur de fin de ce qui est conservé / début à effacer avec **erase**.

```
bool estPair(int i) { return (i%2 == 0); }
```



# Itération sur conteneurs, ranges::views, C++20

## cppitertools, syntaxe « pipe »

Dit « pipe »  
en anglais

- Cppitertools et C++20 introduisent l'opérateur | pour manipuler un conteneur *pendant* l'itération

```
vector v = { 1, 4, 7, 8, 9, 12, 16 };
```

```
using views::reverse, views::filter, views::take; C++20
for (auto&& x : v | reverse | filter(estPair) | take(3))
 cout << x << " ";
cout << endl;
```

Renverse      Garde les valeurs paires      Garde les 3 premières valeurs

Même exemple en cppitertools

```
using namespace iter;
for (auto&& x : v | reversed | filter(estPair) | slice(3))
 cout << x << " ";
cout << endl;
```

```
vector v2 = { 10, 20, 30, 40 };
```

```
for (auto&& [a, b] : zip(v2, v))
 a += b;
```

Quel est le résultat?

Sans copie, à mesure; estPair n'est pas appelé sur toutes les valeurs

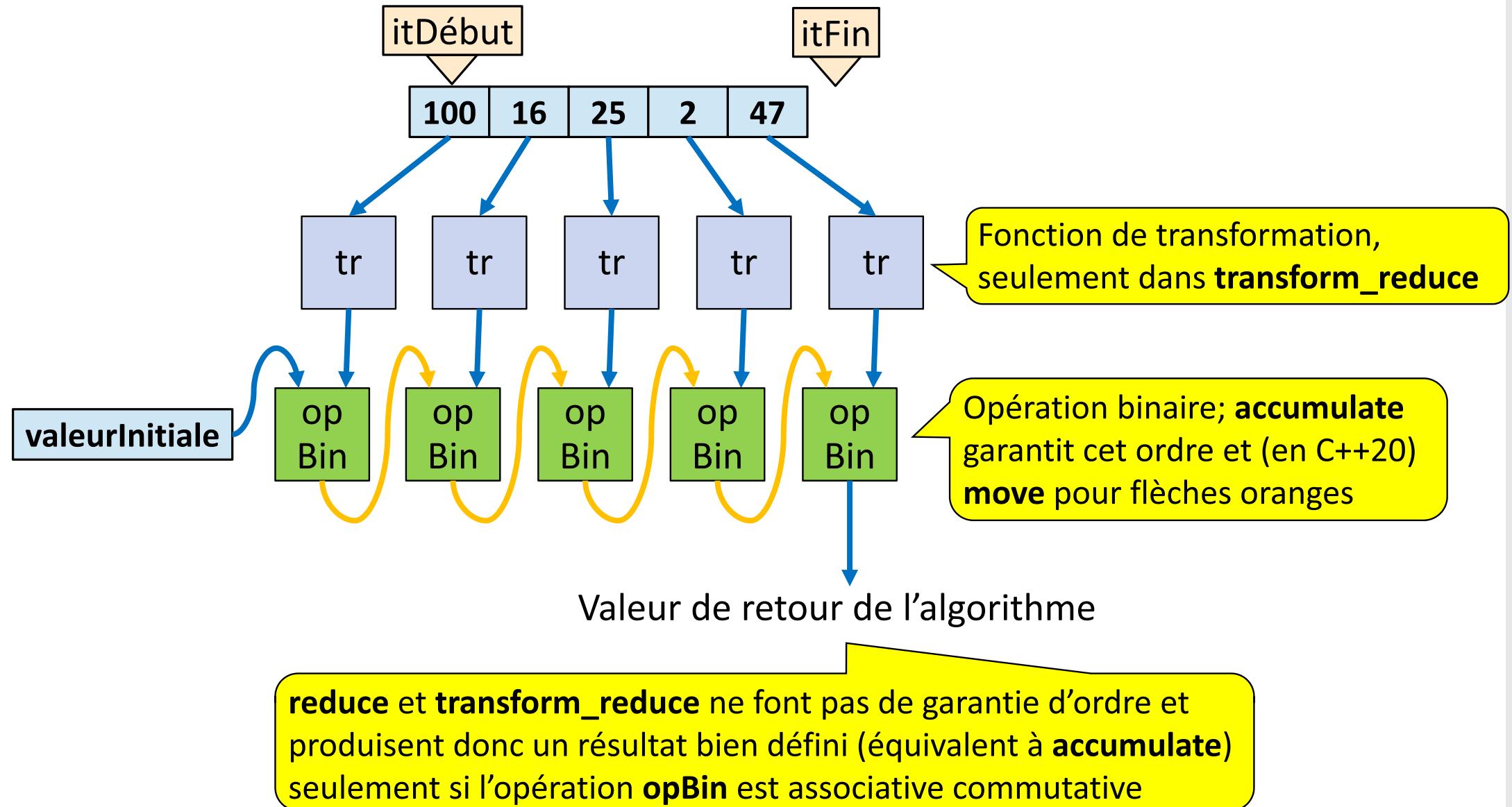
Itération sur plusieurs conteneurs, cppitertools;  
proposé mais pas dans C++20

# Algorithmes pour les valeurs <numeric>

- Comment faire une somme, produit, ou, et?
  - reduce(itDébut, itFin, valeurInitiale, operationBinaire)
    - accumulate si avant C++17 ou opération non associative commutative
    - + par défaut (si paramètre omis)
  - transform\_reduce(itDébut, itFin, valeurInitiale, operationBinaire, transformationUnaire)
    - C++17, comme reduce mais applique transformation avant
- Exemples :
  - int produit = reduce(v.begin(), v.end(), 1, multiplies{});
    - Instanciation d'un type foncteur prédéfini dans <functional>
  - int produit = reduce(v.begin(), v.end(), 1, [](int a, int b) { return a\*b; });

# Algorithmes pour les valeurs <numeric>

## accumulate, reduce, transform\_reduce



# Foncteur: objet fonction « à la main » vs lambda

- Plusieurs conteneurs ou algorithmes demandent un foncteur
- Une expression lambda est souvent suffisante

```
• map<string,double,decltype(
 [](const string& a, const string& b){ return a > b; }
)> uneMap;
```

decltype est le type de l'expression, utilisable comme un type dans une déclaration

C++20, le type d'une lambda sans capture est constructible par défaut

- Mais lambda ne peut pas
  - Faire une surcharge globale de **hash** (spécialisation de template) pour notre type
  - Avoir des méthodes d'accès à son état
  - Avoir un constructeur par défaut qui donne un état

# Foncteur: exemple avec état accessible

```

template <typename T> Classe foncteur template
struct Sommer { Constructeur pour l'état initial
 Sommer(const T& valeurDepart = {}) : somme_(valeurDepart) {}
 void operator()(const T& ajout) { somme_ += ajout; }
 T somme; Attribut pour l'état
};

int main() {
 vector v = { "Bonjour "s, "les "s, "amis"s };
 Sommer concat = ""s; Initialise le foncteur (construction avec argument)
 concat.somme.reserve(transform_reduce(v.begin(), v.end(),
 size_t(0), plus{}, Opération binaire Fonction de transformation
 [](const string& x) { return x.size(); }));
 Exécute une fonction pour chaque élément
 ranges::for_each(v, ref(concat));
 Le foncteur est pris par valeur; std::ref fait un reference_wrapper
 cout << concat.somme << endl;
 Résultat avec et sans ref?
}

```

## Introduction aux interfaces graphiques

Programmation événementielle,  
« widgets », Modèle-Vue-Contrôleur,  
Qt: QObject, signals/slots ...

# Introduction : Programmation Séquentielle vs. Événementielle

- **Programmation séquentielle**

La séquence des opérations effectuées est prédéterminée. Le programme dicte le moment où l'utilisateur doit entrer de l'information, et le moyen par lequel il peut le faire.

- **Programmation événementielle**

La séquence des opérations effectuées est variable. Le programme réagit aux « actions » de l'utilisateur dont la séquence est à la fois inconnue et imprévisible.

# Programmation Séquentielle

- En programmation séquentielle, le programme interroge directement l'utilisateur. S'il désire utiliser le programme, l'utilisateur doit se laisser « contrôler ».
- Exemple :

```
int val1, val2;
cout << "Entrer un premier nombre :" << endl;
cin >> val1;
cout << "Entrer un deuxième nombre :" << endl;
cin >> val2;
cout << "La moyenne est de " << (val1 + val2)/2.0;
```

## Programmation Séquentielle (suite)

- En programmation séquentielle, on ne traite qu'**une seule source d'entrées** à la fois (par exemple le clavier).
- Or, avec le développement d'interfaces graphiques complexes, les sources d'entrées se sont multipliées (clavier, souris, stylet, manette, écran tactile, caméras, etc.)
- La programmation séquentielle n'est pas appropriée à l'utilisation d'interfaces complexes puisqu'elle devrait connaître d'avance le moment et la provenance des entrées.

# Événement (définition)

- En français :

Fait qui survient à un moment donné.

Wiktionsnaire déf.1

Tout ce qui se produit, arrive ou apparaît.

Larousse.fr déf.1

- En informatique :

Un événement est un message indiquant que quelque chose s'est produit. Celui-ci peut, ou non, engendrer une série d'opérations visant à réagir au changement d'état.

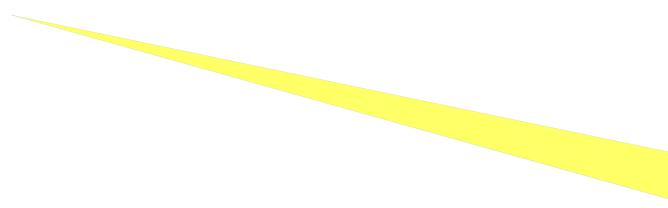
# Programmation événementielle

- La programmation événementielle consiste à écrire un programme qui répond (ou réagit) aux différents événements à mesure que ceux-ci surviennent dans le système.
- Un événement peut être par exemple :
  - Un clic de souris
  - Une touche (ou une combinaison de touches) enfoncée sur le clavier
  - La réception d'un paquet réseau
  - L'arrivée à zéro d'un compteur à rebours (*timer*).

# Gestion des événements

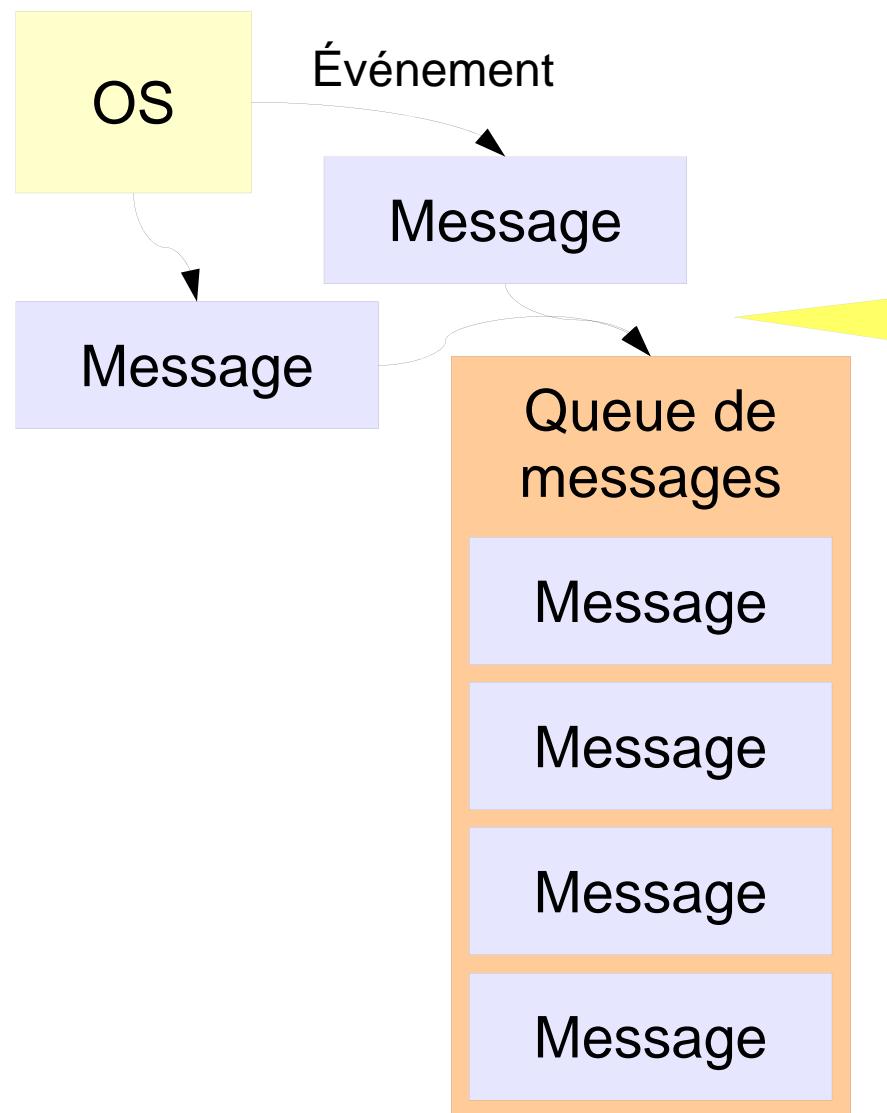
OS

Événement



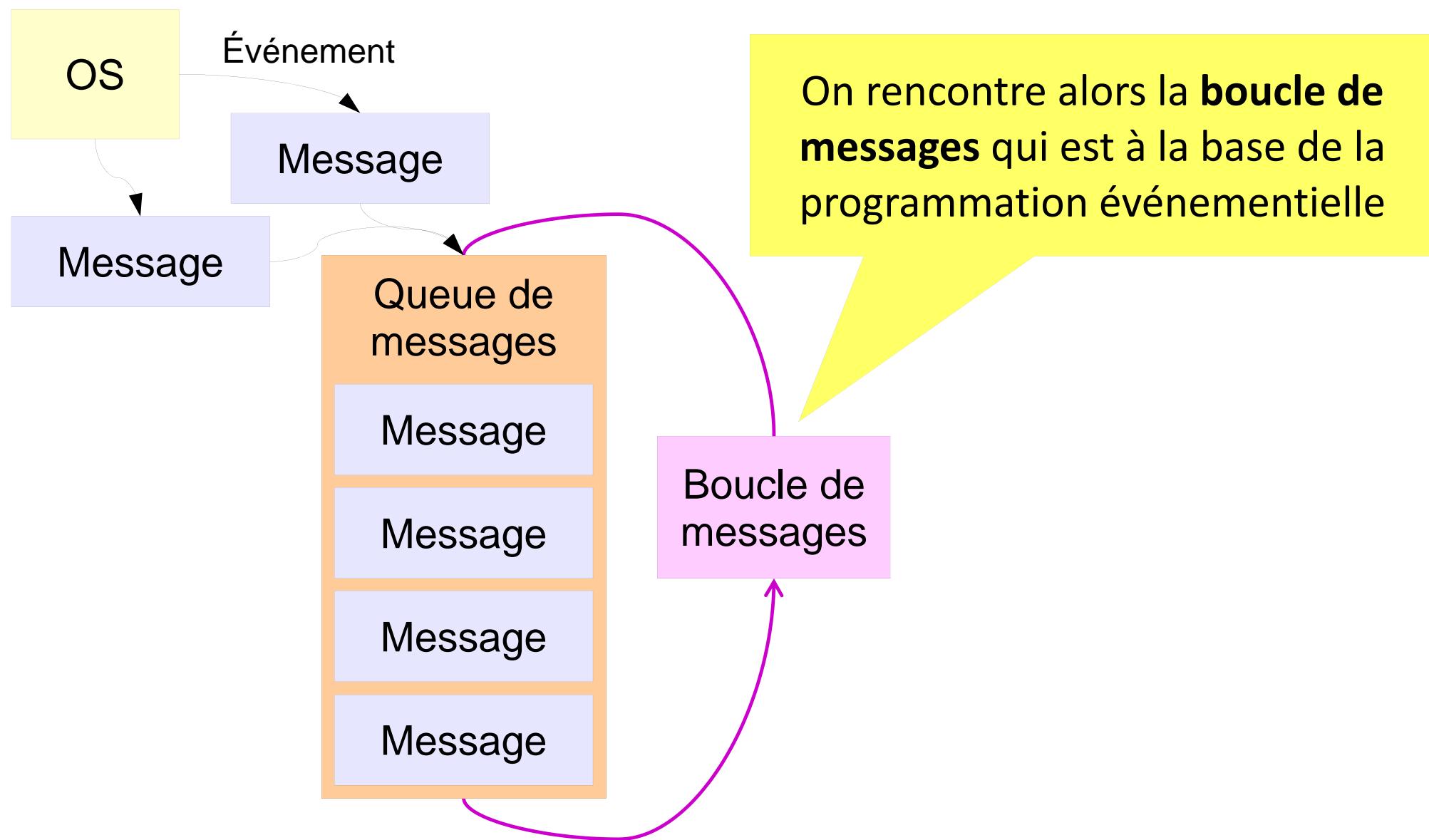
Le **système d'exploitation** est le premier à être informé des différents événements.

# Gestion des événements

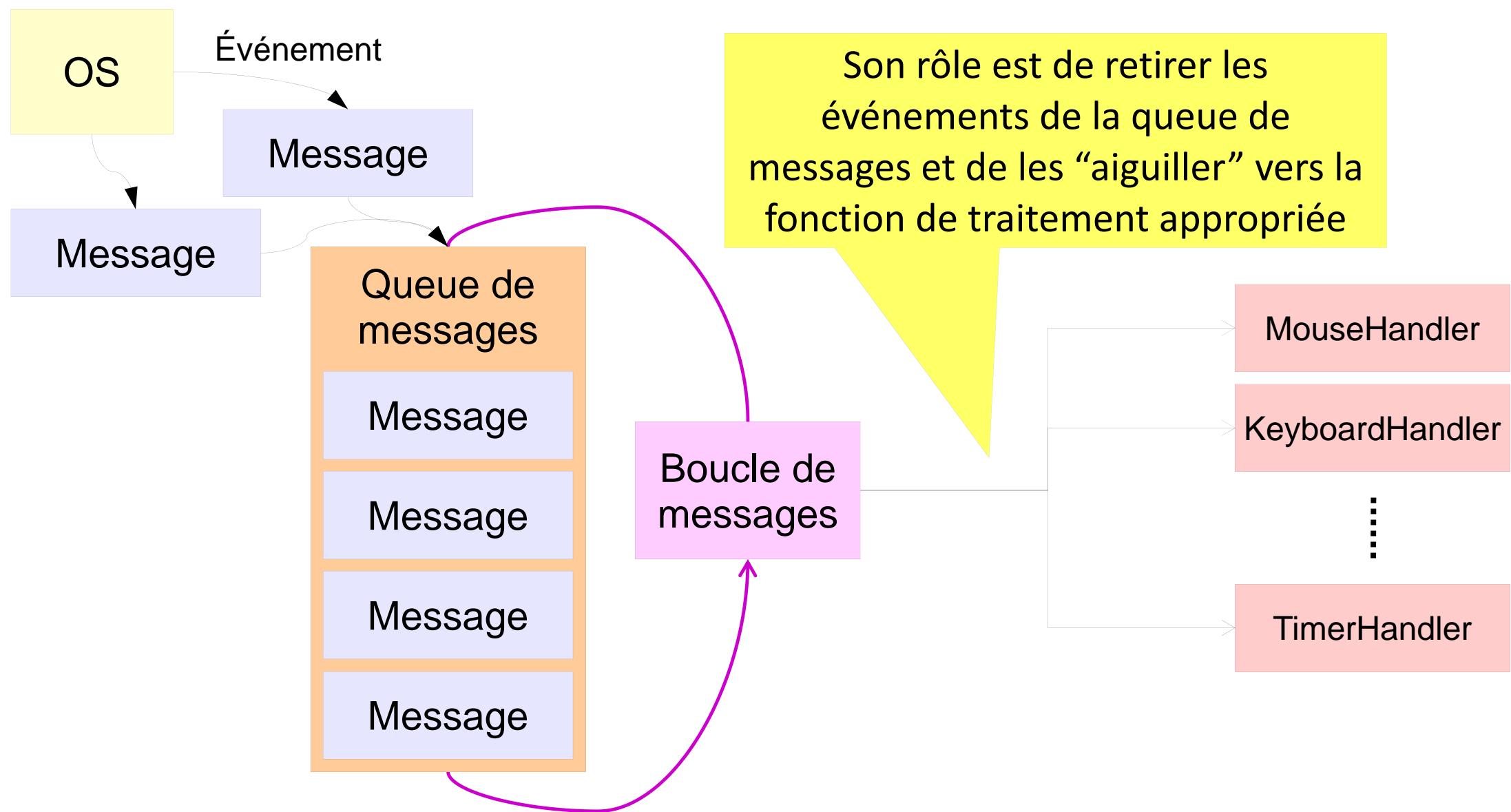


Lorsqu'il reçoit un événement, il l'ajoute dans la file de messages du « programme » approprié (habituellement, celui actif au moment de l'événement, ou celui sous le pointeur de la souris).

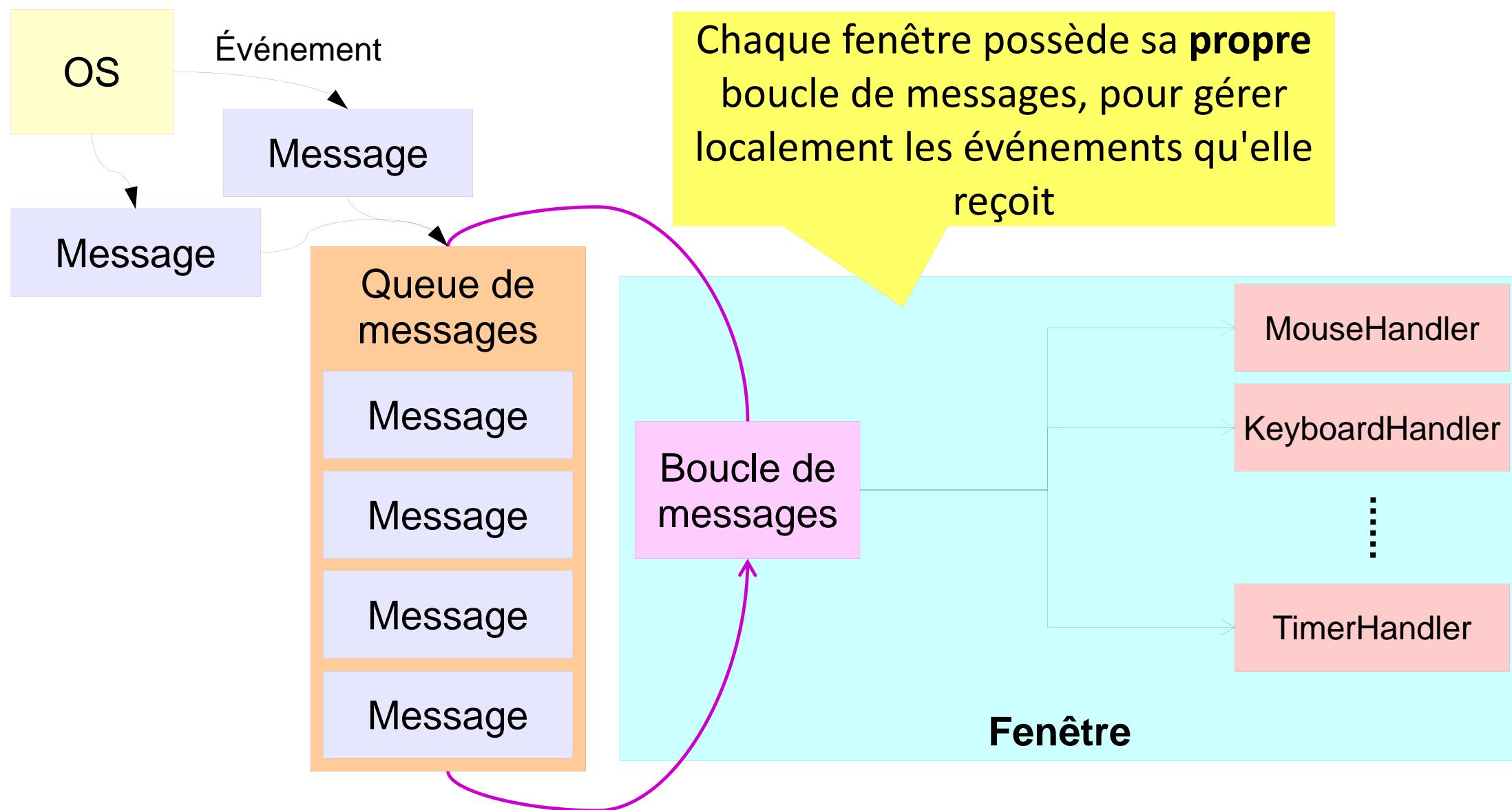
# Gestion des événements



# Gestion des événements

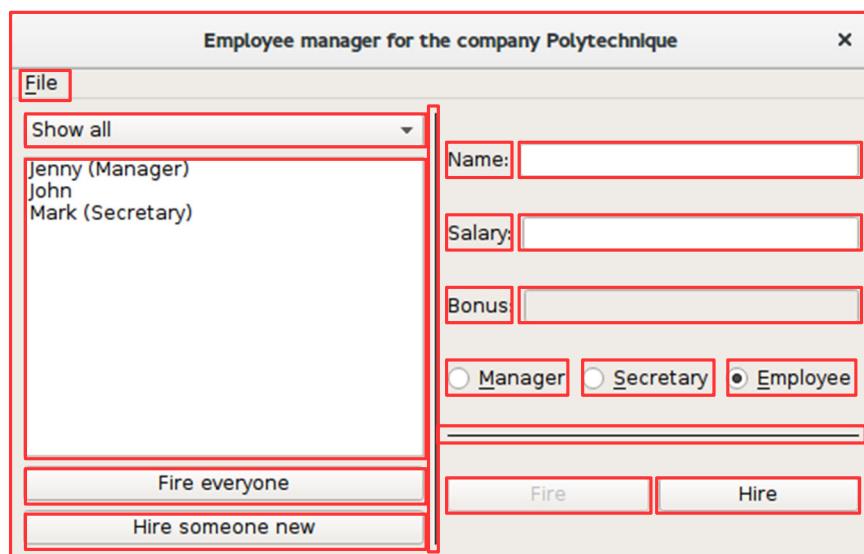


# Gestion des événements



# Éléments d'interface graphique (widgets)

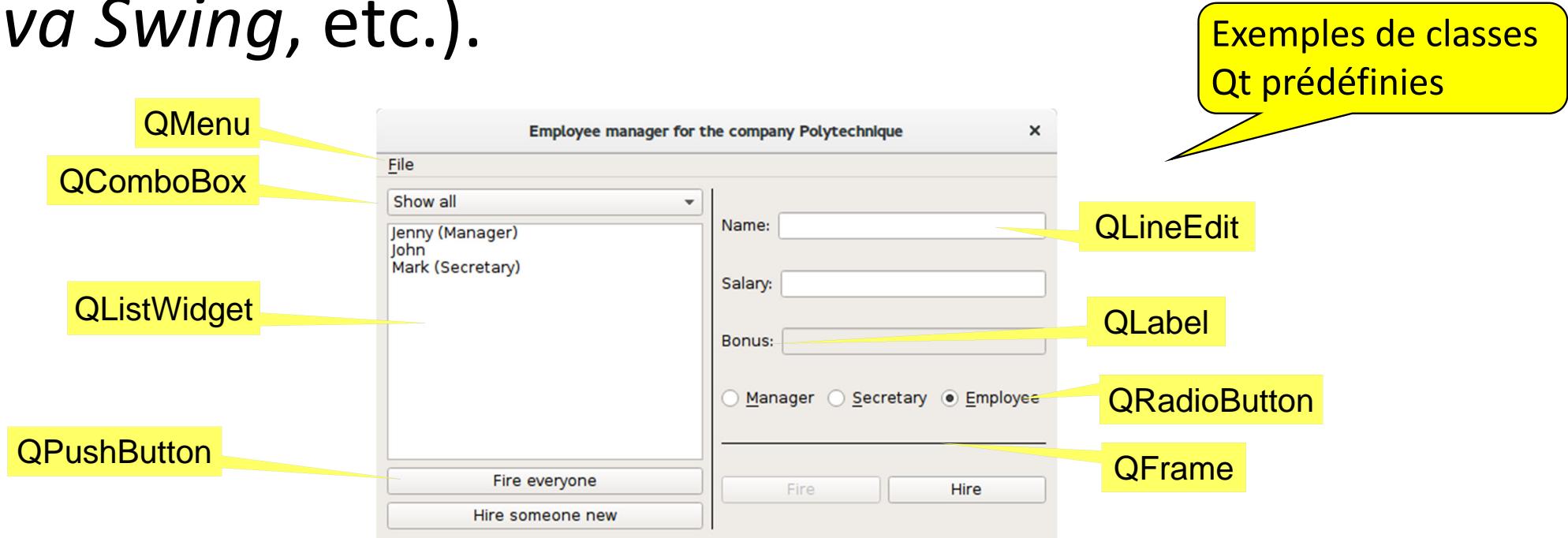
- Tous les éléments d'une interface graphique sont des fenêtres (ou sous-fenêtres). La façon dont ceux-ci réagissent aux différents événements (boucle de messages) dictera leur utilité.



Tous les éléments encadrés en rouge, entre autres, sont conceptuellement des fenêtres d'interface graphique

# Éléments d'interface graphique (widgets)

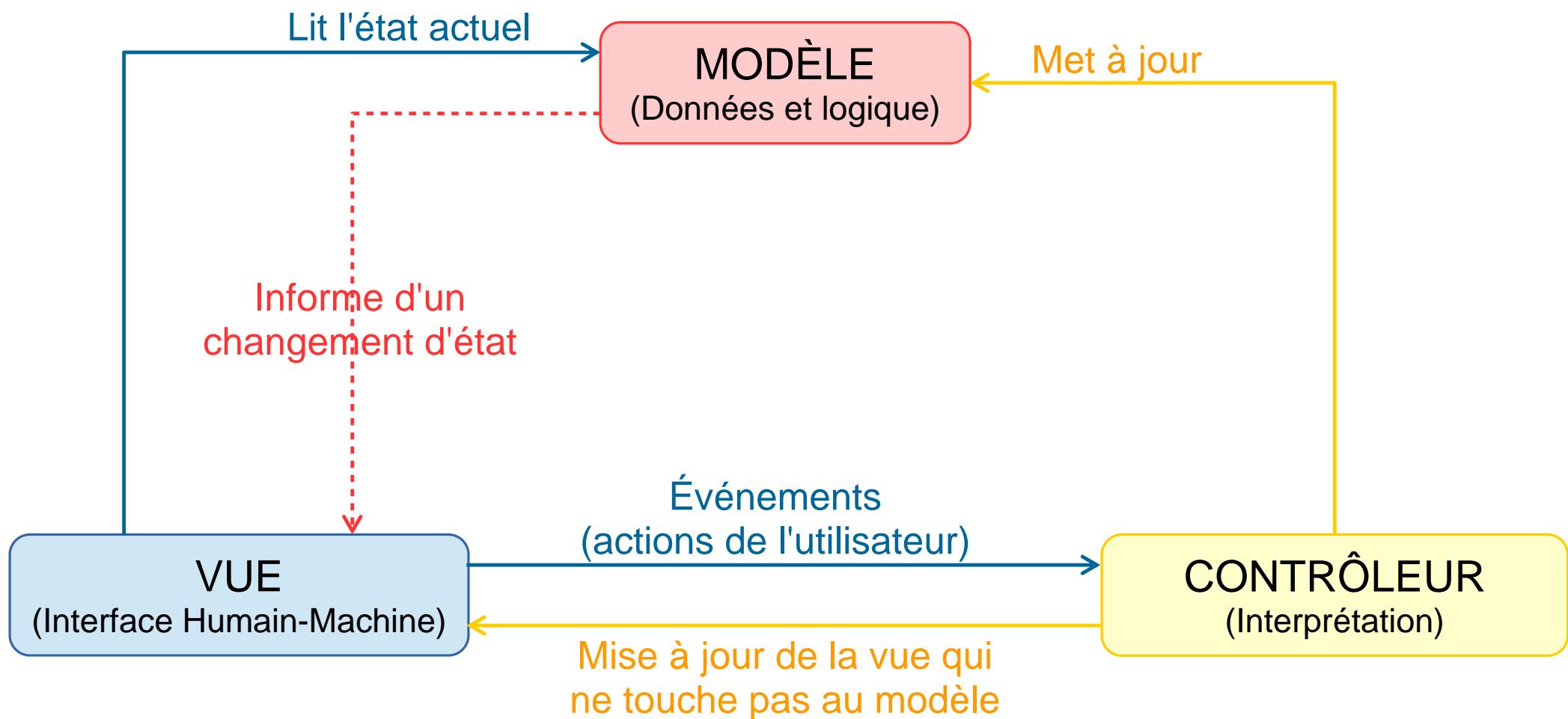
- Certains éléments (*widgets*) sont tellement fréquemment utilisés (boutons, champs texte, etc.) qu'ils sont intégrés aux bibliothèques de développement (*Qt*, *WPF*, *Win32 API*, *wxWidgets*, *Java Swing*, etc.).



# Modèle – Vue – Contrôleur (MVC)

- C'est un **patron de conception** très utile en programmation événementielle et surtout dans la conception d'interfaces graphiques (GUI).
- Cela permet de **séparer les tâches en plusieurs classes pour diminuer la complexité** du programme ou du composant, et d'offrir une **grande flexibilité et réutilisabilité** du code.
- Séparation des données (modèle), de la présentation (vue), et des traitements (contrôleur).

# Modèle – Vue – Contrôleur (Relations)

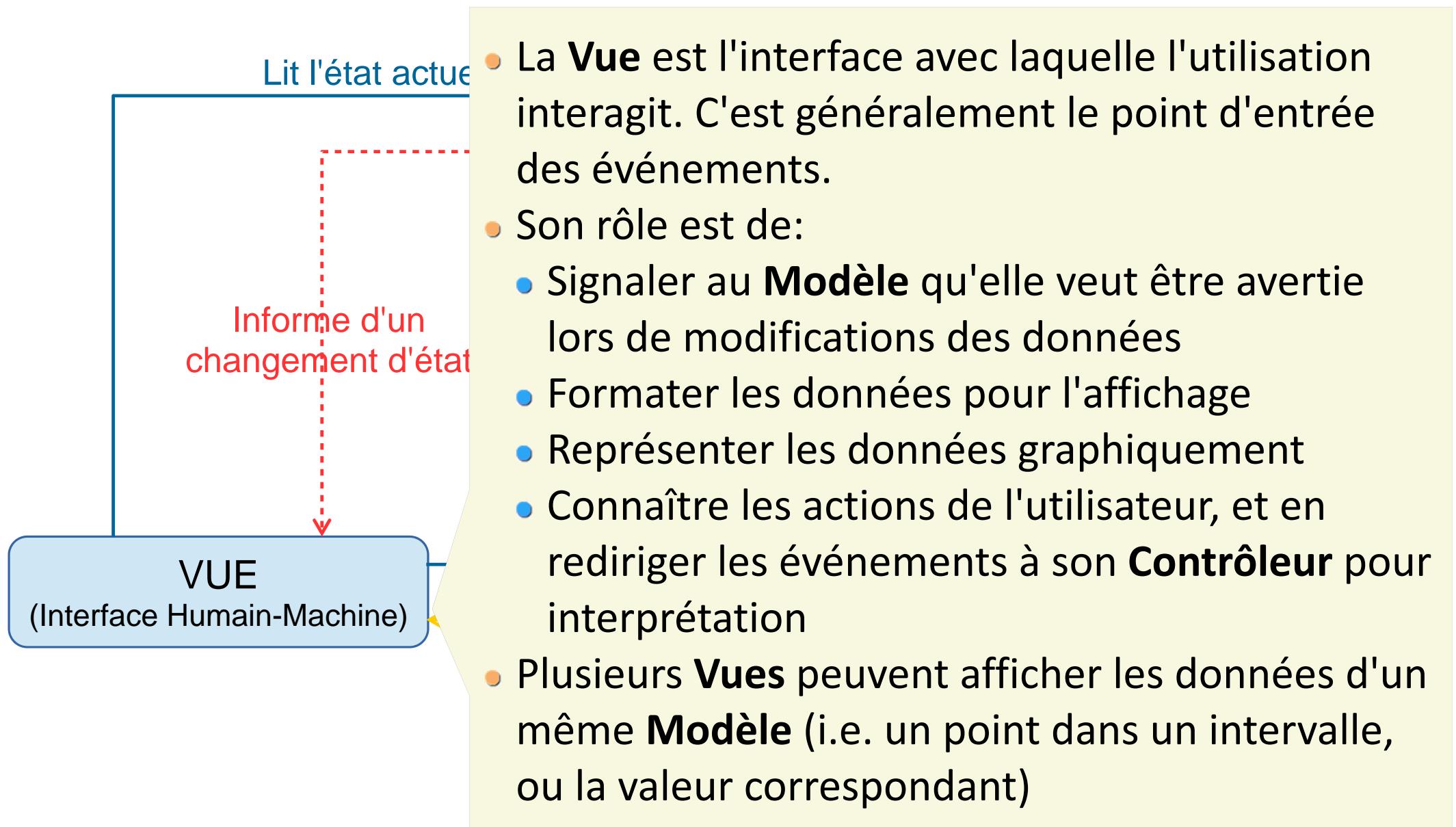


# Modèle – Vue – Contrôleur (Relations)



- Le **Modèle** sert à:
  - Maintenir l'état des données et fournir des méthodes d'accès et de modification pour celles-ci
  - Signaler aux **vues** que l'état de ses attributs a été modifié
  - Gérer les événements qui ne proviennent pas de l'interface graphique (i.e. Timer)
- Le **Modèle** ne formate pas les données pour la sortie: c'est le rôle de la **Vue**.
- Il doit fournir un ensemble de méthodes assez large pour que ses données soient modifiées puisqu'il ne connaît pas le besoin des **Contrôleurs**.

# Modèle – Vue – Contrôleur (Relations)



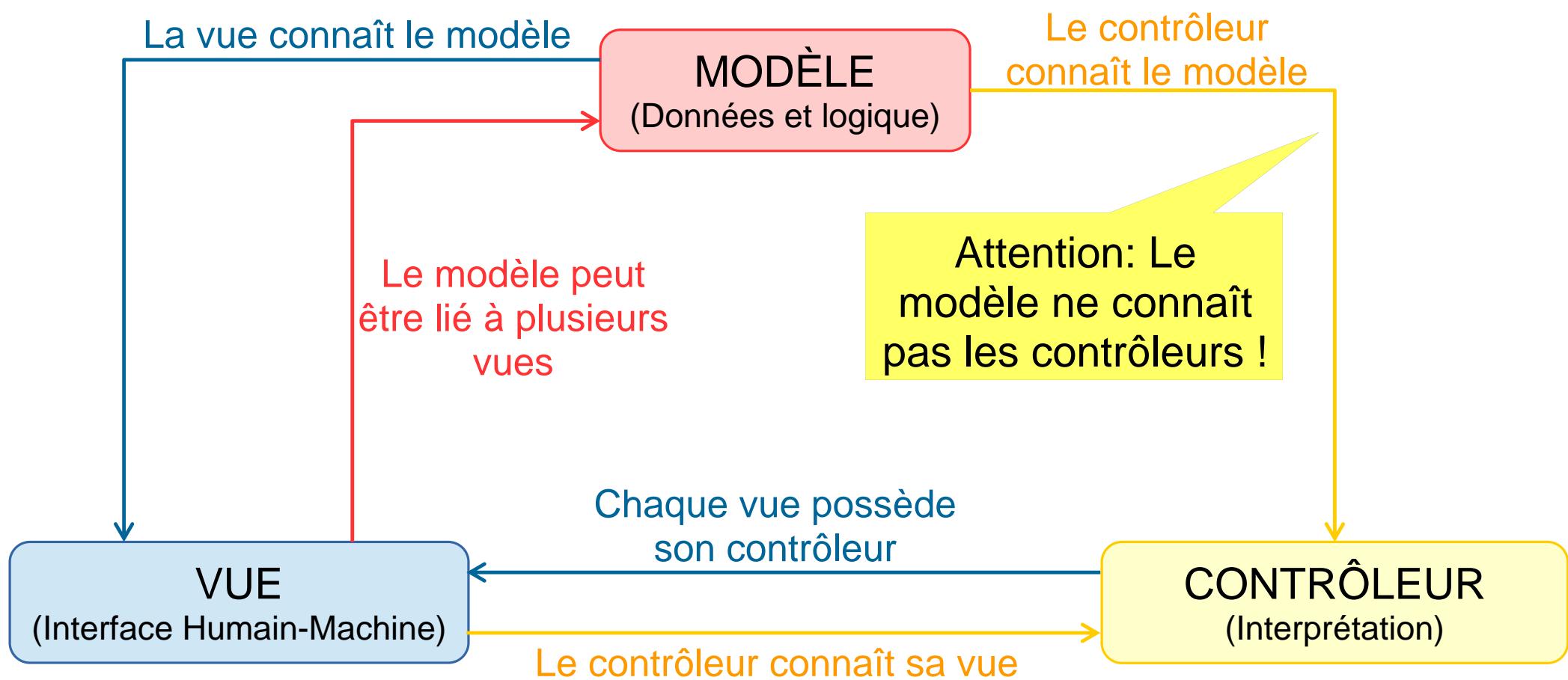
# Modèle – Vue – Contrôleur (Relations)

- Le **Contrôleur** a pour rôle de:
  - Gérer les événements
  - Interpréter les actions de l'utilisateur en un changement sur le **Modèle** ou sur sa **Vue**:
    - Si une donnée doit être modifiée, modifier le **Modèle** (qui le signalera aux **Vues**)
    - Si le changement est purement visuel, l'appliquer à la **Vue** (i.e. couleur d'arrière plan, habillage d'un élément, etc.)
  - Vérifier la validité des actions de l'utilisateur: il ne doit autoriser un changement que s'il est permis à l'utilisateur.
- Le **Contrôleur** ne fait aucun traitement. Il ne modifie pas / n'adapte pas les données. C'est le **Modèle** qui contient la logique sur les données.

Met à jour



# Modèle – Vue – Contrôleur (Liens)



# La bibliothèque graphique Qt : quel est son intérêt ?

- C'est une bibliothèque multiplateformes (GNU/Linux, Windows, Mac OS X, Android, iOS, WinRT, etc.) largement utilisé pour la conception d'applications graphiques.
- **Qt** est notamment connu pour être la bibliothèque sur laquelle repose KDE, l'un des environnements de bureau les plus utilisés dans le monde GNU/Linux.
- Offre environ 500 classes d'éléments graphiques, conteneurs génériques, outils de modification de texte, dessins 2D, etc.
- Simule l'aspect et la convivialité des applications natives des systèmes d'exploitation (*look and feel*).



# La bibliothèque graphique Qt

- Bon, par contre... **Qt** n'utilise pas vraiment MVC, mais une version simplifiée : l'architecture **Modèle-Vue**.



- En fait, dans ce modèle, le **Contrôleur** est intégré à la **Vue**. On met ensemble ce qui interagit avec l'usager, tout en gardant les données séparées de l'affichage.

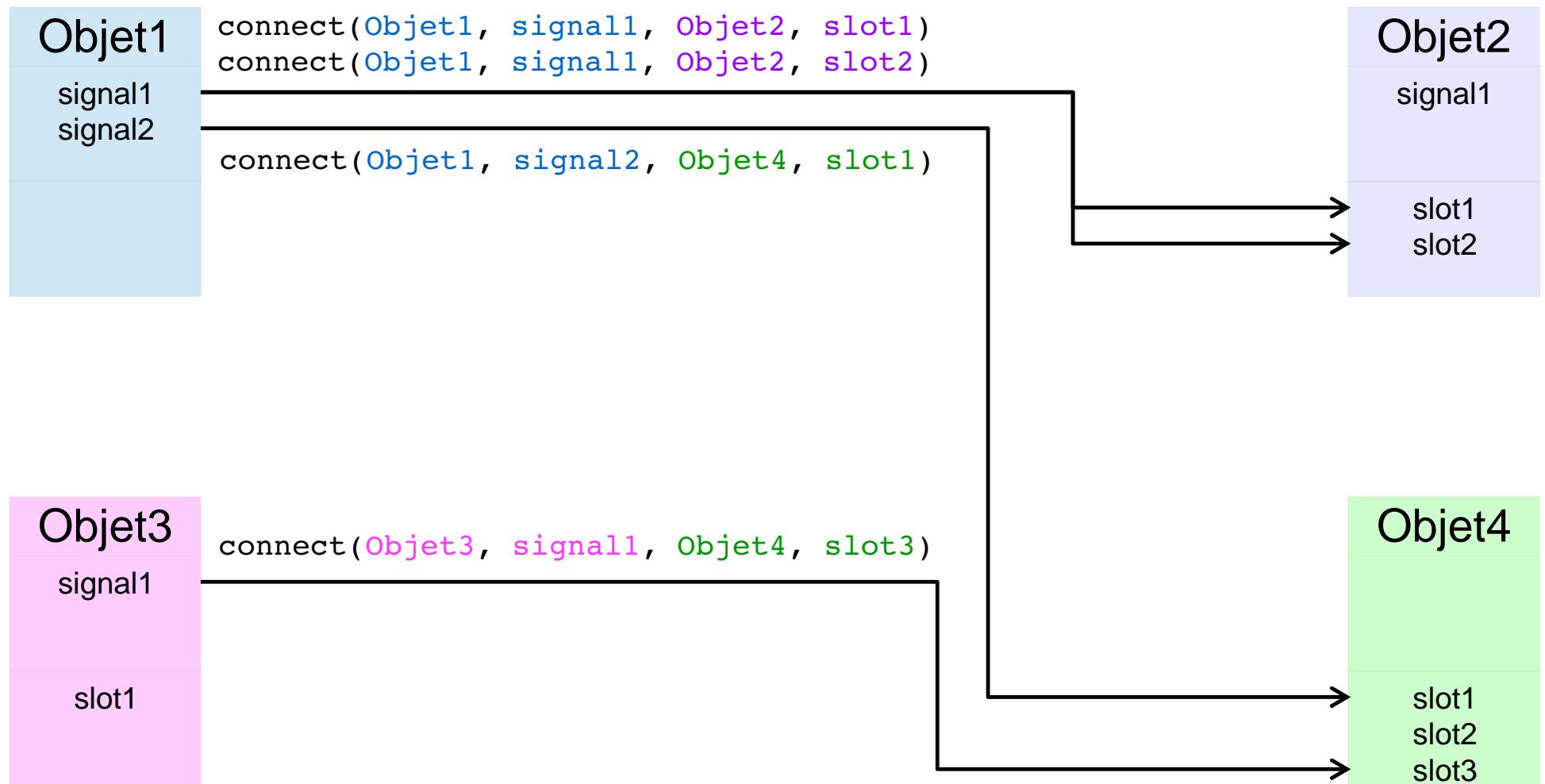
# La bibliothèque graphique Qt

## signals & slots

- Parmi les points forts de **Qt**, on trouve incontestablement le mécanisme de **signals et slots**.
- Ce mécanisme permet de gérer les événements au sein d'une fenêtre en liant une action (i.e. clic sur un bouton) à une réaction (i.e. fermeture de la fenêtre).
- Concrètement :
  - Un **signal** est un message envoyé lorsqu'un événement se produit.
  - Un **slot** est la fonction, généralement une méthode d'une classe, qui est appelée lorsqu'un événement s'est produit.
  - Si on relie un **signal** à un ou plusieurs **slots**, lorsque ce **signal** est émis, le ou les **slots** liés sont exécutés.

# La bibliothèque graphique Qt

## Signals & slots : exemple



# La bibliothèque graphique Qt

## signals & slots : exemple

- Si on considère la classe minimale suivante :

```
class Counter
{
public:
 Counter() { val_ = 0; }
 int getValue() const { return val_; }
 void setValue(int value);

private:
 int val_;
};
```

# La bibliothèque graphique Qt

## signals & slots : exemple

- Une version Qt ressemblerait à ça :

```
#include <QObject>

class Counter : public QObject
{
 Q_OBJECT
public:
 Counter() { val_ = 0; }
 int getValue() const { return val_; }

public slots:
 void setValue(int value);

signals:
 void valueChanged(int newValue);

private:
 int val_;
};
```

On inclut la classe dont on va dériver notre compteur

Notre compteur devient un objet dérivé d'un objet Qt

Cette macro permet de rendre disponibles les signaux et slots

**setValue** est devenu un **slot** : on pourra le lier à un **signal** pour que la valeur change lors de certains événements

On ajoute un **signal** qui sera émis lorsque la valeur changera

# La bibliothèque graphique Qt

## signals & slots : exemple

- Les **signals** n'ont pas d'implémentation, puisque lors de leur émission ce sont des **slots** qui seront exécutés.
- Les **slots**, par contre, doivent donc être implémentés. Pour notre **slot setValue**, on aura donc :

```
void Counter::setValue(int value)
{
 if (value != val_) {
 val_ = value;
 emit valueChanged(value);
 }
}
```

Le mot clé **emit** sert à émettre un **signal**. Ici, on émet le signal **valueChanged** lorsque la valeur est changée, avec pour paramètre la nouvelle valeur.

# La bibliothèque graphique Qt

## signals & slots : exemple

- On peut alors connecter nos **slots** et **signals** ensemble.
- Par exemple, créons deux instances de **Counter**, et lions-les pour que lorsque l'on modifie le compteur **c1**, le compteur **c2** soit modifié, mais pas l'inverse :

```
int main() {
 Counter c1, c2;
 QObject::connect(&c1, SIGNAL(valueChanged(int)),
 &c2, SLOT(setValue(int)));
```

On connecte le **signal valueChanged** de l'objet **c1** au **slot setValue** de l'objet **c2**

```
c1.setValue(42);
cout << "C1: " << c1.getValue()
 << "; C2: " << c2.getValue()
 << endl; // C1: 42; C2: 42
```

Lorsqu'on modifie **c1**, le signal est émis, le slot exécuté, est **c2** est aussi modifié

```
c2.setValue(1337);
cout << "C1: " << c1.getValue()
 << "; C2: " << c2.getValue()
 << endl; // C1: 42; C2: 1337
```

On n'a connecté le signal que dans un sens : rien ne se passe pour **c1** lorsqu'on modifie **c2**

```
}
```

# La bibliothèque graphique Qt

## signals & slots : exemple

- On peut alors connecter nos **slots** et **signals** ensemble.
- Par exemple, créons deux instances de **Counter**, et lions-les pour que lorsque l'on modifie le compteur **c1**, le compteur **c2** soit modifié, mais pas l'inverse :

```
int main() {
 Counter c1, c2;
 QObject::connect(&c1, &Counter::valueChanged,
 &c2, &Counter::setValue);

 c1.setValue(42);
 cout << "C1: " << c1.getValue()
 << ";" C2: " << c2.getValue()
 << endl; // C1: 42; C2: 42

 c2.setValue(1337);
 cout << "C1: " << c1.getValue()
 << ";" C2: " << c2.getValue()
 << endl; // C1: 42; C2: 1337
}
```

Autre écriture valide qui utilise l'adresse de la méthode au lieu des macros **SIGNAL** et **SLOT** fournies par **Qt**

# La bibliothèque graphique Qt

## Hello World, you're so cute.

- Un programme fenêtré peut être écrit en très peu de lignes avec **Qt**, comme le montre cet exemple basique :

Dans **QtWidgets/** (cocher module « widgets »)

```
#include <QApplication>
#include <QLabel>

int main(int argc, char* argv[])
{
 QApplication app(argc, argv);

 QLabel* hello = new QLabel("Hello World, "
 "you're so cute.");
 hello->show();

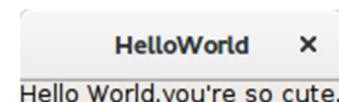
 return app.exec();
}
```

Chaque classe possède un fichier d'entête du même nom

Tout élément de **Qt** qui ne possède pas de parent devient lui-même une fenêtre complète

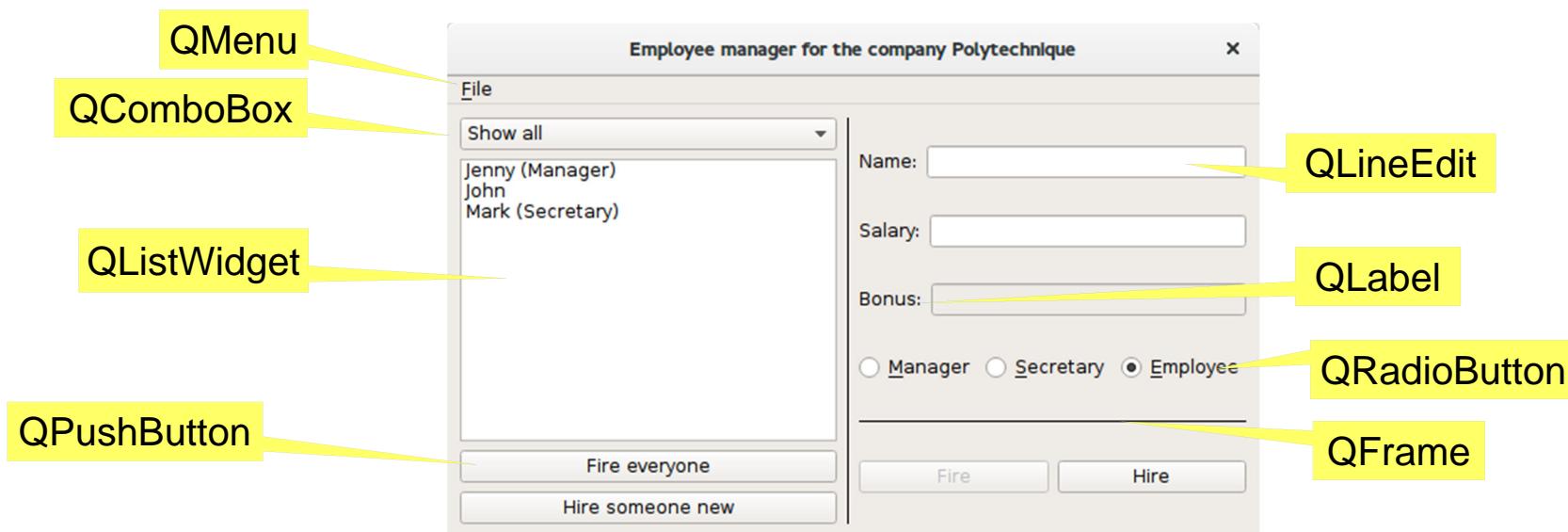
Pour afficher l'élément à l'écran

On démarre la boucle d'événements: le programme ne se terminera que lorsque l'utilisateur aura demandé à quitter



# La bibliothèque graphique Qt Company

- Appliquons l'utilisation de Qt à notre programme de gestion des employés d'une entreprise. On l'a vu, on souhaite obtenir le programme suivant :



- Dans notre cas, le modèle sera la classe **Company**, et la vue sera une nouvelle classe **MainGui**

# La bibliothèque graphique Qt

## Company : la classe Company

- On va tout d'abord ajouter des signaux à notre modèle :

```
class Company : public QObject
{
 Q_OBJECT

/* ... */

signals:
 /**
 * @brief employeeAdded Signal sent when an employee is added
 * @param employee The employee that has been added
 */
 void employeeAdded(Employee* employee);
 /**
 * @brief employeeDeleted Signal sent when an employee is deleted
 * @param employee The employee that has been deleted
 */
 void employeeDeleted(Employee* employee);

/* ... */
};
```

# La bibliothèque graphique Qt

## Company : la classe Company

- Que l'on va émettre lorsque les situations adéquates se présentent :

```
void Company::addEmployee(Employee* employee)
{
 employees_.push_back(employee);
 emit employeeAdded(employee);
}

void Company::delEmployee(Employee* employee)
{
 auto it = find(employees_.begin(), employees_.end(), employee);

 if (it != employees_.end()) {
 Employee* e = *it;
 employees_.erase(it);
 emit employeeDeleted(e);
 }
}
```

# La bibliothèque graphique Qt

## Company : la classe MainGui

- Puis, créer notre nouvelle classe :

```
class MainGui : public QMainWindow
{
 Q_OBJECT
public:
 MainGui(QWidget *parent = 0);
 MainGui(Company* company, QWidget *parent = 0);
 ~MainGui();

private:
 /** @brief setup To setup the GUI */
 void setup();
 /** @brief setMenu To prepare the top menu of the GUI */
 void setMenu();
 /** @brief setUI To prepare the content of the GUI */
 void setUI();
 /** @brief loadEmployees To load the employees of the company into the QListWidget */
 void loadEmployees();
 /** @brief filterHide Function to return whether the given employee has to be hidden */
 void filterHide(Employee*);

/* ... */
```

On fait dériver notre classe de **QMainWindow**, qui dérive de **QWidget**, qui elle-même dérive de **QObject**

Constructeur par défaut qui créera un nouvel objet **Company**

Constructeur par paramètre pour préciser sur quel objet **Company** on veut travailler

# La bibliothèque graphique Qt

## Company : la classe MainGui

- On aura besoin de certains attributs :

```
/* ... */
/** @brief company_ To store the company on which the GUI works */
Company* company_;
/** @brief added_ To store the list of locally created employees */
vector<Employee*> added_;
/** @brief companyIsLocal_ To know if we created a new company locally */
bool companyIsLocal_;
/** @brief currentFilterIndex_ To store the current filter for the employees list */
int currentFilterIndex_;
/** @brief employeesList Graphical list of the employees */
QListWidget* employeesList;
/** @brief .. Graphical edit fields for the employee name/salary/bonus */
QlineEdit *nameEditor, *salaryEditor, *bonusEditor;
/** @brief employeeTypeRadioButtons List of radio buttons for the employee type */
list<QRadioButton*> employeeTypeRadioButtons;
/** @brief fireButton Graphical button to fire an employee */
QPushButton* fireButton;
/** @brief hireButton Graphical button to hire a new employee */
QPushButton* hireButton;
/* ... */
```

Certains attributs servent à stocker des informations pour y accéder plus tard

D'autres vont servir à stocker des éléments graphiques pour pouvoir les utiliser dans différentes méthodes de la classe

# La bibliothèque graphique Qt

## Company : la classe MainGui

- Finalement, on aura des **slots** :

```
/* ... */

public slots:
 /** @brief filterList Slot to filter the list according to the received parameter */
 void filterList(int);
 /** @brief selectEmployee Slot to select an employee given a QListWidgetItem */
 void selectEmployee(QListWidgetItem*);
 /** @brief cleanDisplay To clean the editor on the right of the GUI */
 void cleanDisplay();
 /** @brief changedType To update the editor when we select another type of employee */
 void changedType(int);
 /** @brief fireEveryone To fire all the employees */
 void fireEveryone();
 /** @brief fireSelected To fire only the selected employees */
 void fireSelected();
 /** @brief createEmployee To create a new employee locally */
 void createEmployee();
 /** @brief employeeHasBeenAdded To run when an employee has been added */
 void employeeHasBeenAdded(Employee*);
 /** @brief employeeHasBeenDeleted To run when an employee has been deleted */
 void employeeHasBeenDeleted(Employee*);
};
```

# La bibliothèque graphique Qt

## Company : la classe MainGui

- Et l'implémentation qui correspond :

```
MainGui::MainGui(QWidget *parent) :
 QMainWindow(parent)
{
 company_ = new Company();
 companyIsLocal_ = true;
 setup();
}
MainGui::MainGui(Company* company, QWidget *parent) :
 QMainWindow(parent)
{
 company_ = company;
 companyIsLocal_ = false;
 setup();
}
MainGui::~MainGui() {
 if (companyIsLocal_) {
 delete company_;
 }
 while (!added_.empty()) {
 delete added_.back();
 added_.pop_back();
 }
}
```

On crée un objet **Company** local, et on garde un booléen pour penser à `delete` cet objet à la fin!

Notre objet **Company** contient les employés par agrégation: les employés que l'on va créer par l'interface devront donc être détruits lors de la destruction de notre interface

# La bibliothèque graphique Qt

## Company : la classe MainGui

- **setMenu**, que l'on utilise pour définir le menu en haut de la fenêtre:

```
void MainGui::setMenu() {
 // On crée un bouton 'Exit'
 QAction* exit = new QAction(tr("E&xit"), this);
 // On ajoute un raccourci clavier qui simulera l'appui sur ce bouton (Ctrl+Q)
 exit->setShortcuts(QKeySequence::Quit);
 // On connecte le clic sur ce bouton avec l'action de clore le programme
 connect(exit, SIGNAL(triggered()), this, SLOT(close()));

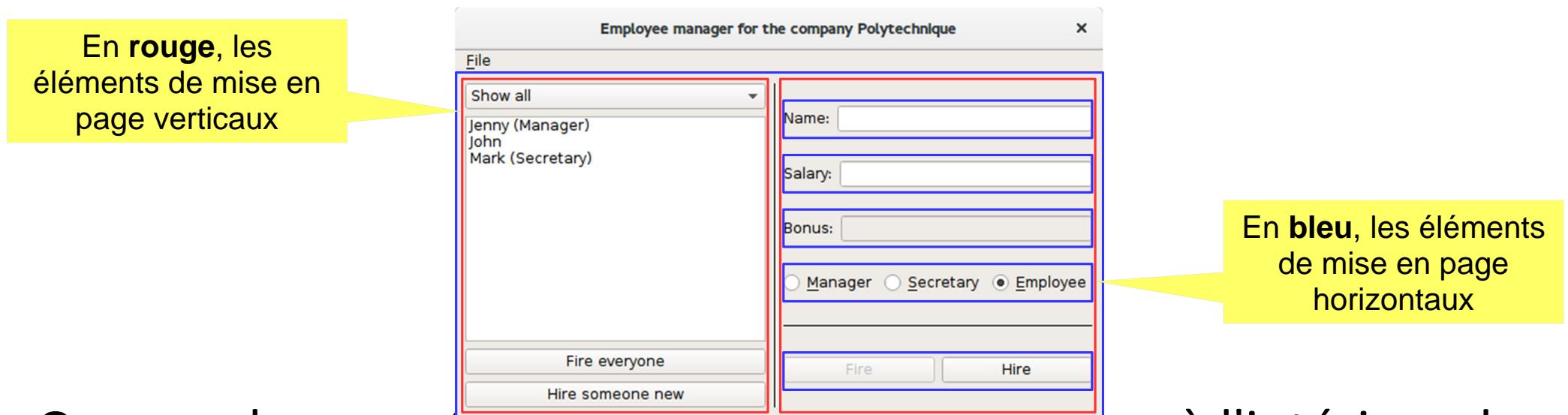
 // On crée un nouveau menu 'File'
 QMenu* fileMenu = menuBar()->addMenu(tr("&File"));
 // Dans lequel on ajoute notre bouton 'Exit'
 fileMenu->addAction(exit);
}
```

- Et **setUI** qui va nous permettre de définir la mise en page interne de notre fenêtre. Mais comment ça marche la mise en page ?

# La bibliothèque graphique Qt

## Company : mise en page

- Pour organiser une application, on a besoin d'utiliser des éléments de mise en page (**Layout**)
- Pour notre application, on utilisera principalement les verticaux et horizontaux (**QVBoxLayout** et **QHBoxLayout**)



- On cumulera ces éléments en les plaçant les uns à l'intérieur des autres pour créer des mises en pages plus complexes.

# La bibliothèque graphique Qt

## Company : mise en page

```

void MainGui::setUI() {
 // Le sélecteur pour filtrer ce que l'on souhaite dans la liste
 QComboBox* showCombobox = new QComboBox(this);
 showCombobox->addItem("Show all"); // Index 0
 showCombobox->addItem("Show only managers"); // Index 1
 showCombobox->addItem("Show only secretaries"); // Index 2
 showCombobox->addItem("Show only other employees"); // Index 3
 connect(showCombobox, SIGNAL(currentIndexChanged(int)),
 this, SLOT(filterList(int)));

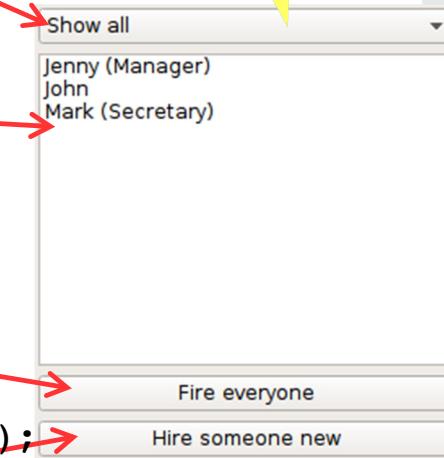
 // La liste des employés
 employeesList = new QListWidget(this);
 employeesList->setSortingEnabled(true);
 connect(employeesList, SIGNAL(itemClicked(QListWidgetItem*)),
 this, SLOT(selectEmployee(QListWidgetItem*)));

 // Le bouton pour congédier tout le monde
 QPushButton* fireEveryoneButton = new QPushButton(this);
 fireEveryoneButton->setText("Fire everyone");
 connect(fireEveryoneButton, SIGNAL(clicked()), this, SLOT(fireEveryone()));

 // Le bouton pour remettre à zéro la vue et créer un nouvel employé
 QPushButton* hireSomeoneButton = new QPushButton(this);
 hireSomeoneButton->setText("Hire someone new");
 connect(hireSomeoneButton, SIGNAL(clicked()), this, SLOT(cleanDisplay()));
 /* ... */
}

```

Lorsqu'on change l'élément sélectionné, le slot **filterList** sera appelé avec pour paramètre le nouvel index



# La bibliothèque graphique Qt

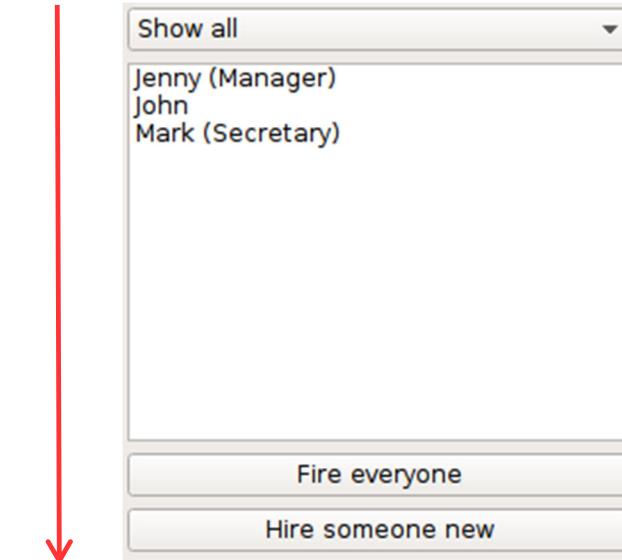
## Company : mise en page

- Puis, on place nos éléments dans un **QVBoxLayout** qui va permettre de les organiser verticalement :

```
/* ... */

// Le premier layout, pour la colonne de gauche, dans lequel on insère les
// éléments que l'on veut dans l'ordre dans lequel on veut qu'ils apparaissent
QVBoxLayout* listLayout = new QVBoxLayout;
listLayout->addWidget(showCombobox);
listLayout->addWidget(employeesList);
listLayout->addWidget(fireEveryoneButton);
listLayout->addWidget(hireSomeoneButton);

/* ... */
```



# La bibliothèque graphique Qt

## Company : mise en page

- On s'occupe maintenant des champs pour entrer les données, que l'on doit organiser dans leurs propres **QHBoxLayout** :

```
/* ... */

// Champ pour le nom
QLabel* nameLabel = new QLabel;
nameLabel->setText("Name:");
nameEditor = new QLineEdit;

QHBoxLayout* nameLayout = new QHBoxLayout;
nameLayout->addWidget(nameLabel);
nameLayout->addWidget(nameEditor);

// Champ pour le salaire
QLabel* salaryLabel = new QLabel;
salaryLabel->setText("Salary:");
salaryEditor = new QLineEdit;

QHBoxLayout* salaryLayout = new QHBoxLayout;
salaryLayout->addWidget(salaryLabel);
salaryLayout->addWidget(salaryEditor);
```



```
// Champ pour le bonus
QLabel* bonusLabel = new QLabel;
bonusLabel->setText("Bonus:");
bonusEditor = new QLineEdit;
bonusEditor->setDisabled(true);

QHBoxLayout* bonusLayout = new QHBoxLayout;
bonusLayout->addWidget(bonusLabel);
bonusLayout->addWidget(bonusEditor);
```

```
/* ... */
```

Désactivé par défaut car le bonus n'est valable que pour un Manager!

**QHBoxLayout:**  
Remplissage de gauche à droite



# La bibliothèque graphique Qt

## Company : mise en page

- On prépare ensuite le groupe de boutons radio pour choisir le type d'employé que l'on veut ajouter :

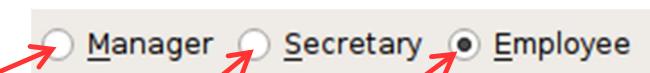
```
QRadioButton* managerRButton = new QRadioButton("&Manager", this);
employeeTypeRadioButtons.push_back(managerRButton);
```

```
QRadioButton* secretaryRButton = new QRadioButton("&Secretary", this);
employeeTypeRadioButtons.push_back(secretaryRButton);
```

```
QRadioButton* employeeRButton = new QRadioButton("&Employee", this);
employeeRButton->setChecked(true);
employeeTypeRadioButtons.push_back(employeeRButton);
```

```
QButtonGroup* employeeTypeButtonGroup = new QButtonGroup;
employeeTypeButtonGroup-> addButton(managerRButton);
employeeTypeButtonGroup-> addButton(secretaryRButton);
employeeTypeButtonGroup-> addButton(employeeRButton);
connect(employeeTypeButtonGroup, SIGNAL(buttonClicked(int)),
 this, SLOT(changedType(int)));
```

```
QHBoxLayout* employeeTypeLayout = new QHBoxLayout;
employeeTypeLayout->addWidget(managerRButton);
employeeTypeLayout->addWidget(secretaryRButton);
employeeTypeLayout->addWidget(employeeRButton);
```



On crée un groupe de boutons qui permet de décocher automatiquement les autres boutons du groupe lorsqu'on en sélectionne un.

C'est aussi ce groupe qui émettra un signal lorsque le type d'employé sera changé: on le connecte à notre slot **changedType**

On place aussi les boutons dans un QHBoxLayout pour les organiser

# La bibliothèque graphique Qt

## Company : mise en page

- On s'occupe enfin des boutons pour congédier/embaucher un employé :

```
// Trait horizontal de séparation
QFrame* horizontalFrameLine = new QFrame;
horizontalFrameLine->setFrameShape(QFrame::HLine);

// Bouton pour congédier la ou les personne(s)
// sélectionnée(s) dans la liste
fireButton = new QPushButton(this);
fireButton->setText("Fire");
fireButton->setEnabled(true);
connect(fireButton, SIGNAL(clicked()),
 this, SLOT(fireSelected()));

// Bouton pour embaucher la personne dont on
// vient d'entrer les informations
hireButton = new QPushButton(this);
hireButton->setText("Hire");
connect(hireButton, SIGNAL(clicked()),
 this, SLOT(createEmployee()));

// Organisation horizontale des boutons
QHBoxLayout* fireHireLayout = new QHBoxLayout;
fireHireLayout->addWidget(fireButton);
fireHireLayout->addWidget(hireButton);
```



Lorsqu'on clique sur ce bouton, le slot **fireSelected** sera appelé



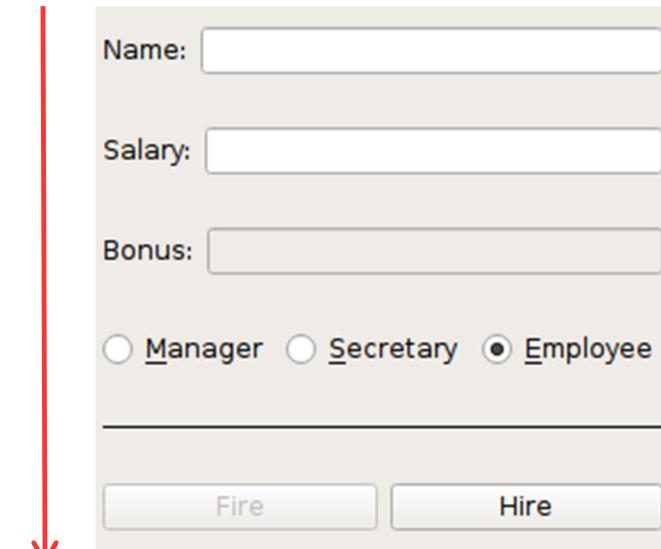
Lorsqu'on clique sur ce bouton, le slot **createEmployee** sera appelé

# La bibliothèque graphique Qt

## Company : mise en page

- On peut finalement placer les éléments de notre colonne de droite dans un **QVBoxLayout** pour les organiser

```
// Organisation pour la colonne de droite
QVBoxLayout* displayLayout = new QVBoxLayout;
displayLayout->addLayout(nameLayout);
displayLayout->addLayout(salaryLayout);
displayLayout->addLayout(bonusLayout);
displayLayout->addLayout(employeeTypeLayout);
displayLayout->addWidget(horizontalFrameLine);
displayLayout->addLayout(fireHireLayout);
```



# La bibliothèque graphique Qt

## Company : mise en page

- Enfin, on s'occupe de l'organisation de la fenêtre principale en y plaçant nos deux **Layout** précédemment préparés :

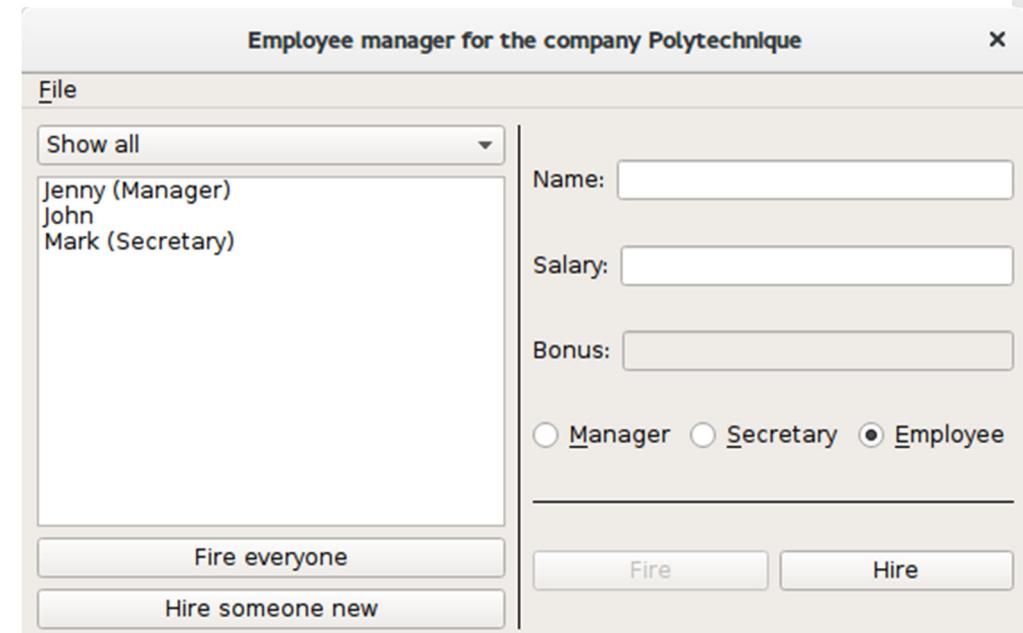
```
// Trait vertical de séparation
QFrame* verticalFrameLine = new QFrame;
verticalFrameLine->setFrameShape(QFrame::VLine);

// Organisation horizontale
QHBoxLayout* mainLayout = new QHBoxLayout;
mainLayout->addLayout(listLayout);
mainLayout->addWidget(verticalFrameLine);
mainLayout->addLayout(displayLayout);

// On crée un nouveau Widget, et on définit son
// layout pour celui que l'on vient de créer
QWidget* widget = new QWidget;
widget->setLayout(mainLayout);

// Puis on définit ce widget comme le widget
// centrale de notre classe
setCentralWidget(widget);

// Enfin, on met à jour le titre de la fenêtre
string title = "Employee manager for the company " + company_->getName();
setWindowTitle(title.c_str());
}
```



# La bibliothèque graphique Qt

## Company : mise en page

- On fera appel à **setMenu** et à **setUI** dans la méthode **setup** à laquelle on fait appel dans nos constructeurs :

```
void MainGui::setup() {
 currentFilterIndex_ = 0;

 // On crée notre menu et notre UI
 setMenu();
 setUI();

 // On connecte les signaux de notre company aux
 // slots créés localement pour agir suite à ces signaux
 connect(company_, SIGNAL(employeeAdded(Employee*)),
 this, SLOT(employeeHasBeenAdded(Employee*)));
 connect(company_, SIGNAL(employeeDeleted(Employee*)),
 this, SLOT(employeeHasBeenDeleted(Employee*)));

 // On charge la liste des employés actuels
 loadEmployees();
}
```

# La bibliothèque graphique Qt

## Company : slots

- Il nous faut maintenant un moyen de charger les employés dans notre liste graphique depuis le modèle :

```
void MainGui::loadEmployees() {
 // On s'assure que la liste est vide
 employeesList->clear();
 // Puis, pour tous les employés dans Company
 int max = company_->getNumberEmployees();
 for (int i = 0; i < max; i++) {
 // On récupère le pointeur vers l'employé
 Employee* employee = company_->getEmployee(i);
 if (employee == nullptr) {
 continue;
 }
 // Et on l'ajoute en tant qu'item de la liste:
 // le nom sera affiché, et le pointeur sera contenu
 QListWidgetItem* item = new QListWidgetItem(
 QString::fromStdString(employee->getName()), employeesList);
 item->setData(Qt::UserRole, QVariant::fromValue<Employee*>(employee));
 item->setHidden(filterHide(employee));
 }
}
```

**QString::fromStdString** permet de transformer un **string** en **QString** pour l'utiliser dans l'interface

**QVariant::fromValue<T>** permet de convertir un élément de type **T** en objet **QVariant**, qui peut être stocké en tant que contenu d'un item d'une liste par exemple

# La bibliothèque graphique Qt

## Company : slots

- Le slot **filterList** permettant de filtrer notre liste lorsqu'on change le type d'employé que l'on souhaite afficher :

```

bool MainGui::filterHide(Employee* employee) {
 switch (currentFilterIndex_) {
 case 1:
 return typeid(*employee) != typeid(Manager));
 case 2:
 return typeid(*employee) != typeid(Secretary));
 case 3:
 return typeid(*employee) != typeid(Employee));
 case 0:
 default:
 return false;
 }
}

void MainGui::filterList(int index) {
 currentFilterIndex_ = index;
 for (int i = 0; i < employeesList->count(); ++i) {
 QListWidgetItem *item = employeesList->item(i);
 Employee* employee = item->data(Qt::UserRole).value<Employee*>();
 item->setHidden(filterHide(employee));
 }
}

```

On récupère le pointeur  
d'Employee contenu dans notre  
item de liste afin de le comparer



# La bibliothèque graphique Qt

## Company : slots

- Le slot **selectEmployee** permettant de mettre dans la colonne de droite les informations de l'employé sélectionné :

```
void MainGui::selectEmployee(QListWidgetItem* item) {
 Employee* employee = item->data(Qt::UserRole).value<Employee*>();
 nameEditor->setDisabled(true);
 nameEditor->setText(QString::fromStdString(employee->getName()));

 salaryEditor->setDisabled(true);
 salaryEditor->setText(QString::number(employee->getSalary()));

 bonusEditor->setDisabled(true);
 if (typeid(*employee) == typeid(Manager)) {
 bonusEditor->setText(
 QString("%1%\n(included in salary)").arg(((Manager*)employee)->getBonus()));
 } else {
 bonusEditor->setText("");
 }
 /* ... */
}
```

Les champs sont désactivés pour la modification, on veut juste afficher les informations ici

Pour un Manager, on affiche le pourcentage, mais le montant est déjà inclus dans le champs salaire

# La bibliothèque graphique Qt

## Company : slots

- Le slot **selectEmployee** permettant de mettre dans la colonne de droite les informations de l'employé sélectionné :

```
/* ... */

list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();
for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {
 (*it)->setDisabled(true);

 bool checked = false;

 if ((typeid(*employee) == typeid(Manager) && (*it)->text().endsWith("Manager")) ||
 (typeid(*employee) == typeid(Secretary) && (*it)->text().endsWith("Secretary")) ||
 (typeid(*employee) == typeid(Employee) && (*it)->text().endsWith("Employee")))
 checked = true;
}

(*it)->setChecked(checked);
}

fireButton->setDisabled(false);
hireButton->setDisabled(true);
}
```

On parcourt tous les boutons radio, on ne veut checker que celui qui correspond

Si le type de notre **Employee** correspond au bouton, on indique donc qu'on doit le checker !

On peut aussi désactiver le bouton pour **embaucher**, et activer le bouton pour  **congédier**

# La bibliothèque graphique Qt

## Company : slots

- Le slot **cleanDisplay** permettant de restaurer la vue initiale :

```

void MainGui::cleanDisplay() {
 nameEditor->setDisabled(false);
 nameEditor->setText(" ");

 salaryEditor->setDisabled(false);
 salaryEditor->setText(" ");

 bonusEditor->setDisabled(true);
 bonusEditor->setText(" ");

 list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();
 for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {
 (*it)->setDisabled(false);
 if ((*it)->text().endsWith("Employee")) {
 (*it)->setChecked(true);
 }
 }

 employeesList->clearSelection();
 fireButton->setDisabled(true);
 hireButton->setDisabled(false);

 nameEditor->setFocus();
}

```

On vide les champs texte, et on les réactive pour l'écriture

On parcourt les boutons radio pour sélectionner celui pour créer un employé simple

On désélectionne tout ce qui était sélectionné dans la liste des employés

On active le bouton pour embaucher, et on désactive le bouton pour congédier

On met le curseur dans le champs pour éditer le nom, afin que l'utilisateur puisse directement écrire!

# La bibliothèque graphique Qt

## Company : slots

- Le slot **changedType** lorsqu'on sélectionne un type d'employé :

```
void MainGui::changedType(int index) {
 if (index == -2) {
 bonusEditor->setDisabled(false);
 } else {
 bonusEditor->setDisabled(true);
 }
}
```

On veut n'activer le champs pour entrer le bonus que si c'est un **Manager** qui est sélectionné dans les boutons radio. L'index **-2** correspond à cette situation dans notre cas!

- Le slot **fireSelected** lorsqu'on clique sur le bouton correspondant :

```
void MainGui::fireSelected() {
 vector<Employee*> toDelete;
 for (QListWidgetItem *item : employeesList->selectedItems()) {
 toDelete.push_back(item->data(Qt::UserRole).value<Employee*>());
 }

 for (Employee* e : toDelete) {
 company_->delEmployee(e);
 }
}
```

On fait appel à **delEmployee** de **Company** pour chaque élément de notre **QListWidget**. On les a mis dans un vecteur au préalable car la suppression des employés va modifier la **QlistWidget**. On fait donc cette suppression en deux étapes.

# La bibliothèque graphique Qt

## Company : slots

- Le slot **fireEveryone** est identique à **fireSelected**, excepté pour le fait que l'on place dans le vecteur tous les items de la liste:

```
void MainGui::fireEveryone() {
 vector<Employee*> toDelete;
 for (int i = 0; i < employeesList->count(); ++i) {
 QListWidgetItem *item = employeesList->item(i);
 toDelete.push_back(item->data(Qt::UserRole).value<Employee*>());
 }

 for (Employee* e : toDelete) {
 company_->delEmployee(e);
 }
}
```

L'écriture de la boucle **for** est différente car on ne peut pas itérer directement sur un **QListWidget**



# La bibliothèque graphique Qt

## Company : slots

- On veut aussi créer un nouvel employé lorsqu'on clique sur **Hire**, le slot **createEmployee** est là pour ça :

```
void MainGui::createEmployee() {
 // On va créer un nouvel employé que l'on placera dans ce pointeur
 Employee* newEmployee;

 // On crée le bon type d'employé selon le cas
 QRadioButton* selectedType = 0;
 list<QRadioButton*>::iterator end = employeeTypeRadioButtons.end();
 for (auto it = employeeTypeRadioButtons.begin(); it != end; it++) {
 if ((*it)->isChecked()) {
 selectedType = *it;
 break;
 }
 }
 /* ... */
}
```



On passe les boutons radio un par un jusqu'à rencontrer celui qui est coché. Dès qu'on l'a trouvé, on saura quel type d'employé on cherche à créer!

# La bibliothèque graphique Qt

## Company : slots

- On veut aussi créer un nouvel employé lorsqu'on clique sur **Hire**, le slot **createEmployee** est là pour ça :

```

/* ... */

// On crée le bon type d'employé selon le cas
if (selectedType->text().endsWith("Manager")) {
 newEmployee = new Manager(nameEditor->text().toStdString(),
 salaryEditor->text().toDouble(),
 bonusEditor->text().toDouble());
} else if (selectedType->text().endsWith("Secretary")) {
 newEmployee = new Secretary(nameEditor->text().toStdString(),
 salaryEditor->text().toDouble());
} else {
 newEmployee = new Employee(nameEditor->text().toStdString(),
 salaryEditor->text().toDouble());
}

// On ajoute le nouvel employé créé à la company
company_->addEmployee(newEmployee);
// Mais on le stocke aussi localement pour pouvoir le supprimer plus tard
added_.push_back(newEmployee);
}

```

**toStdString()** et **toDouble()** permettent respectivement de passer d'un **QString** à un string et un double

On appelle la méthode **addEmployee** de **Company** pour ajouter notre employé nouvellement créé

# La bibliothèque graphique Qt

## Company : slots

- Enfin, on veut réagir lorsqu'un employé est ajouté dans la **Company** sur laquelle on travaille :

```
void MainGui::employeeHasBeenAdded(Employee* employee) {
 // On ajoute le nouvel employé comme item de la QListWidget
 QListWidgetItem* item = new QListWidgetItem(
 QString::fromStdString(employee->getName()), employeesList);
 item->setData(Qt::UserRole, QVariant::fromValue<Employee*>(employee));

 // On change la visibilité de notre nouvel employé selon
 // le filtre actuel
 item->setHidden(filterHide(employee));
}
```

# La bibliothèque graphique Qt

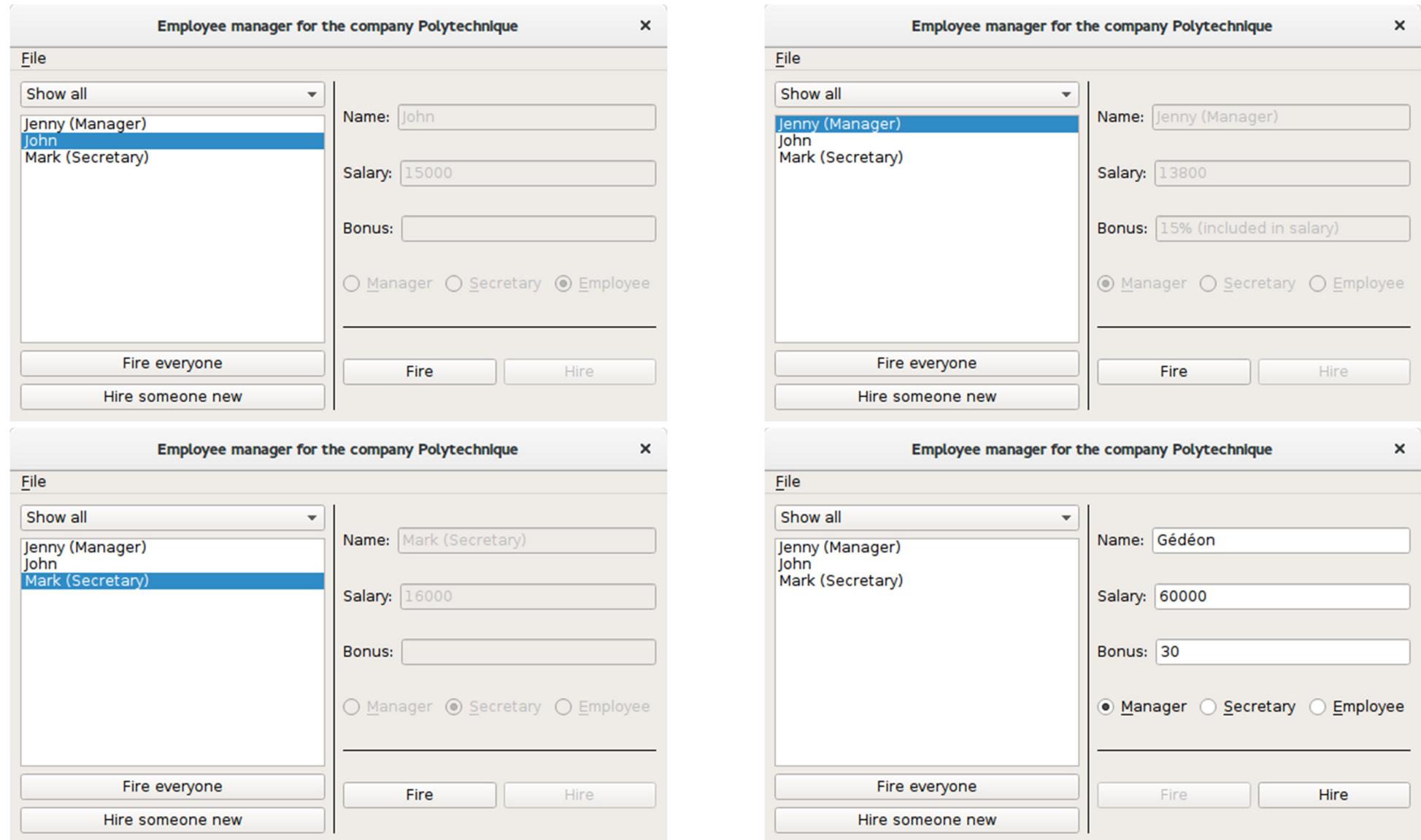
## Company : slots

- ... et lorsqu'un employé est supprimé de la **Company** sur laquelle on travaille :

```
void MainGui::employeeHasBeenDeleted(Employee* e) {
 // On cherche dans notre QListWidget l'employé pour lequel le
 // signal a été envoyé, afin de l'en retirer
 for (int i = 0; i < employeesList->count(); ++i) {
 QListWidgetItem *item = employeesList->item(i);
 Employee* employee = item->data(Qt::UserRole).value<Employee*>();
 if (employee == e) {
 // delete sur un QListWidgetItem va automatiquement le retirer de la liste
 delete item;
 // Si l'employé faisait partie de ceux créés localement, on veut le supprimer.
 auto it = std::find(added_.begin(), added_.end(), e);
 if (it != added_.end()) {
 delete *it;
 added_.erase(it);
 }
 // L'employé ne devrait être qu'une fois dans la liste...
 break;
 }
 }
 // On remet à zéro l'affichage de la colonne de gauche étant
 // donné que les employés sélectionnés ont été supprimés
 cleanDisplay();
}
```

# La bibliothèque graphique Qt

## Company : interface finale



## Exceptions

Assertions, retour d'erreurs,  
types d'exceptions, RAII,  
exceptions dans un constructeur

# Motivation

- Lorsqu'une situation d'erreur se produit, on aimerait:
  - Avertir l'utilisateur qu'une telle situation s'est produite
  - Récupérer la situation, lorsque possible, et continuer l'exécution du programme
- Par exemple, si une situation d'erreur se produit à cause d'une erreur entrée par l'utilisateur, on lui permettra d'en entrer une nouvelle

## Motivation (suite)

- Soit la fonction suivante pour calculer la valeur  $(x+y)/(x-y)$ :

```
double fonction1(double x, double y)
{
 return (x+y) / (x-y);
}
```

- Que devons-nous faire dans la situation où  $x = y$ ?
  - Un système où double suit la norme IEEE 754 donnera +infini s'ils sont positifs, -infini ni négatifs et NAN si zéros.

# Assertions

- Si on ne veut pas traiter les  $\pm\infty$ , une solution consiste à utiliser une assertion, qui vérifie que  $x$  est différent de  $y$ :

```
double fonction1(double x, double y)
```

```
{
```

```
 assert(x != y);
```

```
 // ou: Expects(x != y);
```

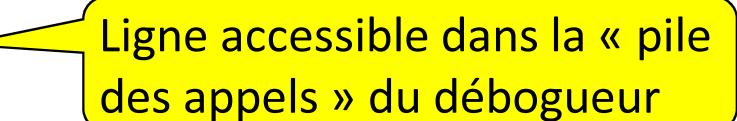
```
 return ((x+y)/(x-y));
```

```
}
```

Si l'expression est fausse, le programme s'arrête avec un message d'erreur qui indique le numéro de ligne

assert est dans <cassert>  
GSL propose Expects

## Assertions (suite)

- Si l'expression à l'intérieur du assert/Expects est vraie, on continue l'exécution du programme
- Si elle est fausse, le programme s'arrête; assert: affiche le nom du fichier, le numéro de la ligne et l'expression; Expects: *terminate*   
Ligne accessible dans la « pile des appels » du débogueur
- **utile pour le débogage** et peut aider la lisibilité en indiquant une contrainte
- On n'utilise pas cette méthode pour afficher un message d'erreur à l'utilisateur, elle ne permet pas de récupérer la situation, lorsque c'est possible

# Interruption du programme

- Une autre solution consiste à afficher un message et interrompre l'exécution du programme:

```
double fonction1(double x, double y)
{
 if (x == y) {
 cout << "Erreur: division par zéro\n";
 exit(-1);
 }
 return ((x+y)/(x-y));
}
```

Interrupt l'exécution du programme et envoie la valeur -1 au programme qui a lancé ce programme.

Toute valeur != 0 est considérée une erreur.

Visual Studio affiche « Sortie de [...]. Code : -1. »

- Cette solution ne permet toujours pas de récupérer la situation lorsque possible

## Retour d'une valeur

- On pourrait utiliser une valeur de retour booléenne/entière/... pour indiquer une erreur ou un type d'erreur
- Dans ce cas, la fonction appelante devra décider ce qu'elle fera
- Dans notre exemple, cette approche pose un problème, puisque la fonction retourne déjà une valeur

## Retour d'une valeur (suite)

- Une solution consisterait à mettre le résultat dans une variable passée par référence:

```
bool fonction1(double x, double y, double& resultat)
{
 if (x == y)
 return false;

 resultat = (x+y) / (x-y);
 return true; ← Retourne vrai si correct
}
```

- On n'a plus une belle fonction, mais une procédure qui retourne son résultat par effet de bord. À éviter!

## Retour d'une valeur (suite)

- On pourrait plutôt retourner une paire qui contient le résultat et la valeur booléenne:

```
pair<double, bool> fonction1(double x, double y) {
 if (x == y)
 return { 0.0, false };
 return { (x+y) / (x-y), true };
}
```

- Ou utiliser std::optional

```
optional<double> fonction1(double x, double y) {
 if (x == y)
 return {};
 return (x+y) / (x-y);
}
```

C++17

Peut ne pas avoir de valeur;  
similaire à permettre None  
en Python

Pas de valeur

Une valeur de type double

## Retour d'une valeur (suite)

- Le retour d'une valeur spéciale (bool ou optional) alourdit l'implémentation de l'appel; chaque appel doit tester

```
if (auto&& [resultat, estBon] = fonction1(x, y); estBon)
 cout << resultat << endl;
else cout << "Erreur\n";
if (auto resultat = fonction1(x, y)) Une bonne solution si c'est fréquent
 cout << *resultat << endl; Version avec optional
else cout << "Erreur\n"; * prend la valeur (n'est pas un pointeur)
```

- Si c'est une situation exceptionnelle, on perd du temps à l'exécution et rend le programme plus difficile à lire pour son exécution normale.

# Exceptions

- Ce qu'il nous faut est une solution qui n'exige pas de changer la valeur de retour de la fonction
- ... et qui fait une **séparation claire entre le “bon chemin”** (good path) et le **“mauvais chemin”** (bad path)
- Cette solution doit par contre **interrompre le cours normal de l'exécution, lorsqu'une situation exceptionnelle** se présente, et indiquer l'erreur qui pourra être traitée par une des fonctions appelantes
- Le C++ a un *mécanisme de traitement d'exception*
  - Pour les cas exceptionnels
  - Souvent avec une implémentation dite « zero overhead »

Pas comme Python où les exceptions sont utilisées aussi dans les cas fréquents

Aucune instruction ajoutée dans l'exécution normale pour traiter les exceptions; tout mis au moment de l'exception.

# Exceptions (suite)

- Le principe est simple: lorsqu'une situation exceptionnelle se présente, on lance une exception:

```
double fonction1(double x, double y)
```

```
{
```

```
 if (x == y)
```

L'exception est un objet; la classe standard **logic\_error** (dans `<stdexcept>`) a un constructeur qui prend un message.

```
 throw logic_error("Division par zero");
```

Interrompt la fonction,  
propage l'exception

Devrait définir notre propre classe dérivée  
(exemple plus loin) au lieu d'un type standard  
(Core Guidelines E.14)

```
 return (x+y) / (x-y);
```

```
}
```

# Exceptions (suite)

- Lorsqu'une exception est lancée, l'exécution normale est interrompue
- **On cherche un gestionnaire pour l'exception qui vient d'être lancée**
  - Débute la recherche dans le bloc qui contient le **throw**
  - Remonte dans la fonction appelante
  - Puis son appelante...
  - Jusqu'à ce qu'on trouve une fonction qui sait traiter l'exception
- **Si aucune fonction ne traite cette exception, le programme se termine abruptement**

Le débogueur permet de voir où l'exception a été lancée

# Exceptions (suite)

- Pour pouvoir traiter une exception, le ***throw*** doit être (indirectement dans une fonction appelante) dans un **bloc *try***
- Le gestionnaire d'exception est indiqué par ***catch***
- Comme plusieurs types d'exception peuvent se produire, on peut avoir plus d'un bloc ***catch*** pour un même bloc ***try***
- On exécute le **premier bloc *catch* qui accepte le type d'exception généré**
  - On ***catch* toujours par référence**, pour permettre le polymorphisme

catch par valeur est permis si on a un type d'exception sans héritage (pas std::exception)

# Exceptions – exemple

- Soit la fonction appelante suivante:

```
void fonction2()
{
 ifstream fichier("paires_de_valeurs.txt");
 while (resteDesDonneesDans(fichier)) {
 double x, y;
 fichier >> x >> y;
 cout << fonction1(x, y) << endl;
 }
}
```

Boucle infinie s'il y a une erreur de lecture

Avec la version simple de fonction1: affiche par exemple « inf » si  $x == y$  pas 0, « -nan(ind) » si  $x$  et  $y$  sont 0...

```
// Avec:
bool resteDesDonneesDans(istream& fichier) {
 return !(fichier.eof() || ws(fichier).eof());
}
```

Si on n'est pas ( à la fin de fichier ou à la fin après avoir sauté les espaces )

# Exceptions – exemple (suite)

- Avec le traitement d'exception version 1:

```
void fonction2_1()
{
 ifstream fichier("paires_de_valeurs.txt");
 try {
 fichier.exceptions(ios::failbit); Active les exceptions sur fail
 while (resteDesDonneesDans(fichier)) {
 double x, y;
 fichier >> x >> y;
 cout << fonction1(x, y) << endl;
 }
 } catch (exception& e) { On traite ici les exceptions de la classe exception et
 cout << "Erreur: " classes dérivées, dont logic_error et ios::failure
 << e.what() what() retourne le message
 << endl; d'erreur de l'exception
 }
}
```

Sur exception  
on tente de  
l'attraper ici

Exécute la fin de fonction après le traitement d'exception (ne continue pas la boucle)

Peut-être trop générique

# Exceptions – exemple (suite)

- Avec le traitement d'exception version 2:

```

void fonction2_2()
{
 ifstream fichier("paires_de_valeurs.txt");
 try {
 fichier.exceptions(ios::failbit);
 while (resteDesDonneesDans(fichier)) {
 try {
 double x, y;
 fichier >> x >> y;
 cout << fonction1(x, y) << endl;
 }
 catch (logic_error& e) {
 cout << "Valeurs incorrectes: " << e.what() << endl;
 }
 }
 }
 catch (ios::failure& e) {
 cout << "Erreur de lecture: " << e.what() << endl;
 }
}

```

**Sur exception on tente de l'attraper ici**

**On traite ici les exceptions de la classe **logic\_error** et classes dérivées**

**Continue la boucle après, si on a attrapé ici**

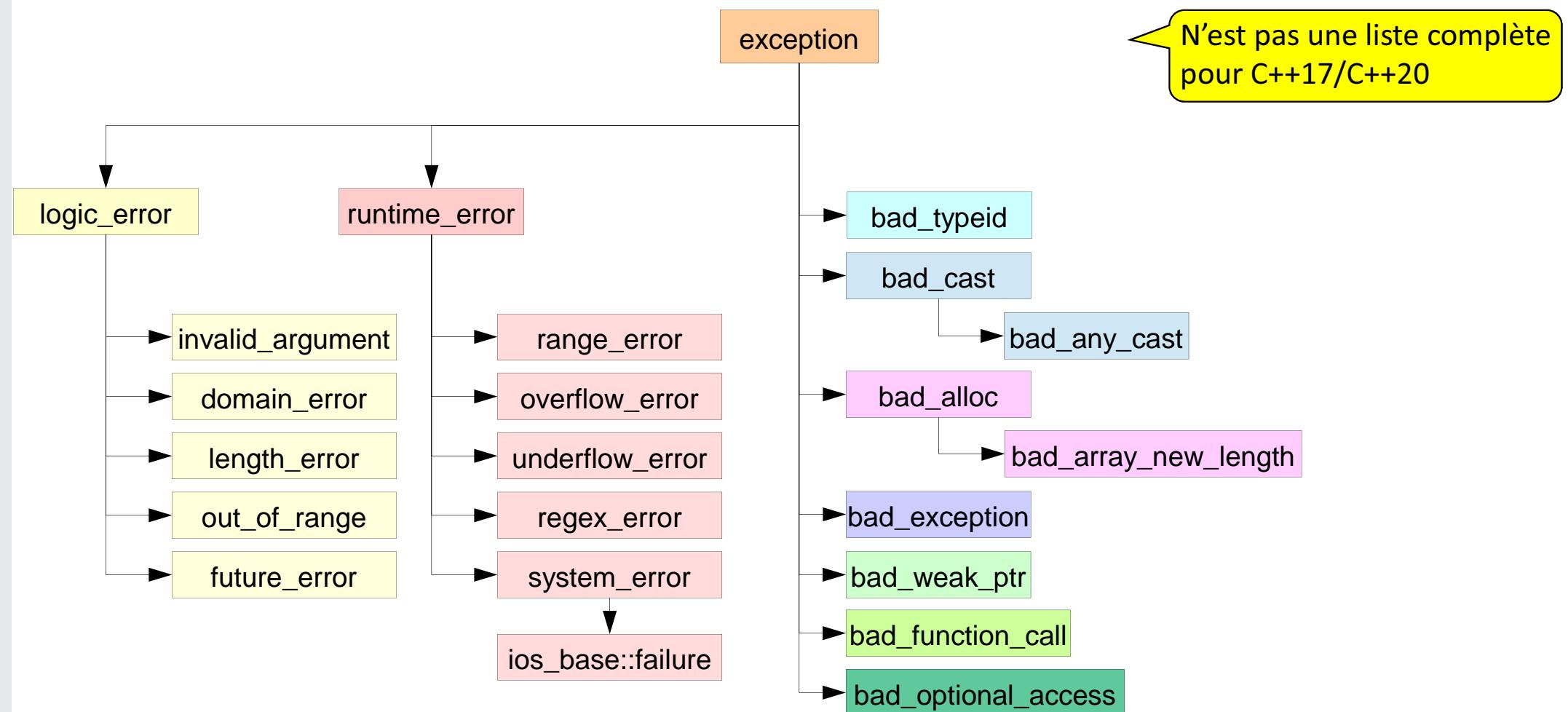
**Ici on traite les **fail** sur le fichier**

# Types d'exception

- Une exception lancée peut être de n'importe quel type, en autant qu'il y ait un type correspondant dans un catch
- En général, on lance **une exception d'un classe définie par le programmeur, qui hérite d'une classe de la bibliothèque C++**
- Attention: le nom de l'exception devrait être lié au problème conceptuel, pas au code qui a causé l'exception (mais le message peut être spécifique)

# Types d'exception (suite)

- Les exceptions en C++ (dans `<stdexcept>`):



## Types d'exception (suite)

- Soit une fonction qui déclare un bloc **try**
- Il n'est pas nécessaire que cette fonction définisse un **catch** pour tous les types d'exception qui peuvent se produire
- Si une exception non traitée par cette fonction a été lancée, on continuera de remonter dans les fonctions appelantes jusqu'à ce qu'on en trouve une qui la traite

# Hiérarchie de classes pour les exceptions

- Il est avantageux de définir une hiérarchie de classes pour les exceptions
- En effet, en utilisant une référence dans le **catch**, on peut profiter du polymorphisme
- Par exemple, si une exception lancée est du type **invalid\_argument**, elle sera traitée si dans l'argument du catch on a une référence à un objet de la class **logic\_error**

# Types d'exception – exemples

```
class InvalidValueException : public logic_error
{
public:
 using logic_error::logic_error;
};
```

On déclare une classe d'exception qui dérive d'une classe prédéfinie de C++.

Hérite des constructeurs, au lieu d'avoir à définir notre propre constructeur. (C++11)

```
class EmployeeNotFoundException : public runtime_error
{
public:
 using runtime_error::runtime_error;
};
```

# Types d'exception – exemple (suite)

```
try
{
 ...
 bool ok;
 double salary = salaryEditor->text().toDouble(&ok);
 if (!ok) {
 throw InvalidValueException("The salary of the
 employee is not in a correct format");
 } Lance l'exception
 ...
}
catch (InvalidValueException& e)
{
 QMessageBox messageBox;
 messageBox.critical(0, "Error when hiring a new employee",
 e.what());
 return;
}
```

Traite les exceptions de type `InvalidValueException` ou dérivées, lancées dans le bloc `try` (ou des fonctions appelées dans ce bloc).

# Types d'exception – exemple (suite)

```

void Company::delEmployee(Employee* employee)
{
 ...
 if (it != employees_.end()) {
 Employee* e = *it;
 employees_.erase(it);
 emit employeeDeleted(e);
 } else {
 throw EmployeeNotFoundException("Could not delete the
 employee as it was not found");
 }
}

```

Dans maingui.cpp:

```

try {
 company_->delEmployee(e);
}
catch (EmployeeNotFoundException& e) {
 QMessageBox messageBox;
 messageBox.critical(0, "Error when firing an employee",
 e.what());
}

```

L'exception lancée dans la fonction membre delEmployee

... sera captée par le bloc catch dans la fonction appelante de maingui.cpp.

# Traitement d'exception par défaut

Trois points

- Une clause **catch (...)** est utilisée pour capter toute exception, peu importe son type
- Ceci est souvent utilisé pour faire du ménage (comme désallouer des pointeurs si pas intelligents) et relancer l'exception
- L'instruction **throw;** est utilisée pour relancer une exception (sans recréer l'objet exception)
- **throw objet;** jette par valeur selon le type statique → pour throw polymorphe: ajouter méthode virtuelle **raise()** à la classe de l'exception, qui fait **throw \*this** dans la version de **raise()** de **chaque classe dérivée**

Sans argument

Possible « slicing »

# Déroulage de la pile d'exécution

- Lorsqu'une exception est lancée, tous les appels de fonction situés entre le point de lancement et le bloc **try** se terminent
- Tous les objets locaux construits dans chacune des fonctions appelées sont détruits en ordre inverse de leur construction
- **Mais les pointeurs ne seront pas désalloués**
- Donc, quand on a alloué un pointeur, il faut **capter l'exception, désallouer le pointeur et relancer l'exception** (ou utiliser des pointeurs intelligents)

# Déroulage de la pile d'exécution exemple

```
Employee* e = nullptr;
try {
 e = new Employee();
 if (e->getNom() == "John")
 {
 // Faire quelque chose.
 }
 delete e;
}
catch (...) {
 delete e;
 throw;
}
```

Commence à **nullptr**: si c'est **new** qui lance l'exception, on veut que **delete e** soit correct.

Si une exception est lancée, il faut désallouer la mémoire avant de la laisser se propager.  
Mais ce n'est pas beau: utiliser les pointeurs intelligents.

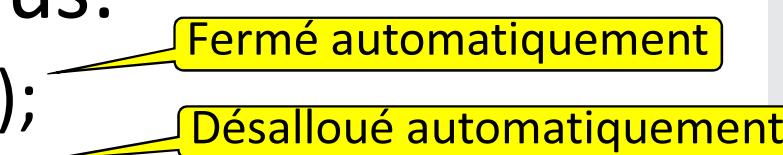
# Déroulage de la pile d'exécution exemple avec pointeur intelligent

```
{
 auto e = make_unique<Employee>();
 if (e->getNom() == "John")
 {
 // Faire quelque chose.
 }
}
```

e est un objet local qui est détruit s'il y a une exception qui sort du bloc où il est construit, et son destructeur s'occupe de la désallocation

# RAII

## « Resource Acquisition Is Initialization »

- La garantie d'appel des destructeurs des objets permet l'idiome RAII
  - Lie l'acquisition à la durée de vie d'un objet
  - La ressource est acquise à l'initialisation d'un objet
  - Elle est libérée par le destructeur de cet objet
- Préférer le RAII aux paires open/close, new/delete ...
  - RAII fait le nettoyage « automatiquement »
  - Particulièrement utile en cas d'exceptions
- Exemples de RAII que nous avons vus:
  - ifstream fichierDonnees("donnees.txt");
  - auto employee = make\_unique<Employee>();

# Exemple RAII

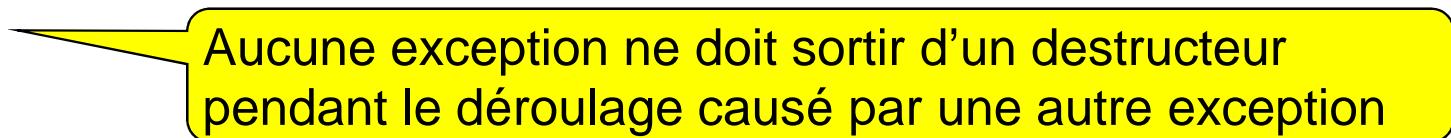
- Pour une certaine raison vous devez utiliser **fopen** de C dans un programme au lieu de fstream.
  - Écrivez un emballage (« wrapper ») RAII C++

```
class File {
public:
 File(const char* filename, const char* mode) :
 file_(fopen(filename, mode))
 {}
 ~File() { fclose(file_); } Destructeur ferme
 FILE* get() { return file_; }
 operator FILE* () { return get(); } Opérateur de conversion implicite permet d'utiliser l'objet comme si c'était un FILE* (le type que retourne normalement fopen)
private:
 FILE* file_;
};

int main() {
 File monFichier("rien.txt", "w"); Fermé automatiquement
 fprintf(monFichier, "Bonjour\n"); Utilise l'opérateur implicite car fprintf prend un FILE*
}
```

Implémentation minimale, on pourrait supporter **string** et avoir des méthodes close/open pour permettre d'ouvrir après la construction et fermer avant la destruction de l'objet

# Exception lancée dans un constructeur

- Si une erreur se produit dans un constructeur, la seule manière de la traiter est de lancer une exception
- Il est important de noter qu'une exception lancée dans un constructeur implique que **l'objet n'a pas été construit**
- **Le destructeur de cette classe ne sera donc pas appelé**
  - Ceux des attributs et classes de bases le sont  Ces objets sont bien construits
- Il faudra donc s'assurer de désallouer tous les pointeurs alloués au moment où une exception est lancée par lui ou un de ses appels
  - → RAI! très avantageux ici
- Note: NE JAMAIS lancer une exception dans un destructeur!  Aucune exception ne doit sortir d'un destructeur pendant le déroulage causé par une autre exception

## Variables et méthodes de classe

« static » avec :  
variables/fonctions globales, variables  
locales, attributs, méthodes

# Mot clé « static »

- A plusieurs significations, selon le contexte
  - Dans tous les cas on dira « statique » en langage parlé
- Les contextes sont:
  - Déclaration de variable ou fonction globale
    - Symbole n'est pas exporté
    - Symbole peut être défini dans plusieurs unités de compilation
  - Déclaration de variable locale
    - Construction/initialisation unique réutilisée d'un appel à l'autre
  - Attribut ou méthode
    - S'applique à la classe et non un objet particulier de cette classe

# Variables et fonctions globales statiques

- Global = déclaré hors des fonctions ou classes
  - Peuvent être dans un **namespace** Autre chapitre; le terme exact est alors « namespace scope »
- Statique = pas visible à l'extérieur du .cpp
  - En C++ un **namespace** sans nom est parfois préférable à **static** dans ce contexte
  - L'édition de lien ne regardera pas le symbole pour le lier
- Si un .hpp fait une telle définition incluse dans plusieurs .cpp
  - la variable ou fonction sera plusieurs fois dans l'exécutable
  - chaque .cpp utilisera sa version
- Ex. de syntaxe: `static int variable;`  
`static int f(int x) { return x + 2; }`

# Variables locale statique

Vu au chap.1 avec variable **const**

- Local = déclaré dans une fonction Dit « block scope » car dans un bloc {} d'une fonction
- Statique = construit/initialisé une seule fois
  - Construite lors de la première exécution de la ligne de définition
  - Pas détruite à la fin du bloc
  - Détruite à la fin du programme, après les variables locales du main, en ordre inverse de construction des variables statiques
- Similaire à une variable globale mais:
  - Visible juste dans une fonction
  - Construite à la première utilisation Utile pour le patron singleton (LOG1410) et le « static initialization order fiasco »

# Variables locale statique (suite)

- Exemple de construction à l'utilisation :

```
MaClasse& getInstanceMaClasse() {
 static MaClasse instance; → Constructeur appelé une seule fois
 return instance; → Retourne cet objet par référence; correct puisqu'il
} → n'est pas détruit à la fin de la fonction
```

- Exemple de fonction avec état (générateur) :

```
uint64_t aleatoire() {
 static uint64_t etat = time(nullptr); → Initialisé lors du premier appel; time
 etat = etat * 25214903917U + 11U; → est appelé une seule fois à ce moment
 return etat; → (Constantes venant de glibc)
}
```

On modifie l'état par rapport à l'état précédent conservé entre les appels

- État est généralement mieux avec une classe (page →)

# Sans variables locale statique

- Foncteur avec état (générateur) :

```
class Aleatoire {
public:
 void germe(uint64_t x) { etat_ = x; }
 uint64_t operator() () {
 etat_ = etat_ * 25214903917U + 11U;
 return etat_;
 }
private:
 uint64_t etat_ = time_(nullptr);
};
Aleatoire aleatoire2;
```

On peut ajuster l'état (ne pouvait pas avec la version var locale static)

L'état est devenu un attribut

Foncteur global; pourrait aussi être local ou construit à l'utilisation.

# Attribut statique ou variable/attribut de classe

- Est partagé par toutes les instances de la classe
  - L'unique instance de l'attribut est **construite au lancement** comme une variable globale
  - **Existe même s'il n'y a pas d'instance de la classe**
  - Ne prend pas d'espace dans l'objet
- Accessible comme tout attribut ( . ou -> ) ou avec Classe::attribut
- Respecte la visibilité (public/protected/private)
- Doit être défini dans le .cpp
  - Comme si c'était une var globale mais avec Classe:: devant  
LeType Classe::attribut = uneValeur;
  - Ou **inline** dans le .hpp 

# Attribut statique – exemples de syntaxe

// Dans le .hpp:

```
class AttributStatique {
private: Mot static dans la déclaration
 static std::string attribut1_;
 static std::string attribut2_;
 inline static std::string attribut3_ = "ijkl"; inline dans la déclaration
};

inline std::string AttributStatique::attribut2_ = "efgh"; inline hors ...
 (C++17)
```

// Dans le .cpp:

```
string AttributStatique::attribut1_ = "abcd";
Pas le mot static dans la définition
```

# Méthode statique ou méthode de classe

- Est ce qui s'applique pas sur une instance de la classe
  - Elle n'a **pas d'objet this**
  - → peut accéder aux attributs/méthodes **static** ou celles non static d'autres instances
    - Similairement à une fonction globale **friend**
- Accessible comme toute méthode ( `.` ou `->` ) ou avec `Classe::méthode`
- Exemples:

```
int x = numeric_limits<int>::max();
```

Méthode statique qui donne la valeur max du type  
(constante connue à la compilation)

```
QObject::connect(bouton, &QPushButton::clicked, maFonction);
```

Méthode statique pour connecter un signal Qt

# Méthode statique – exemple

- Compteur d'instances

```
// Dans Employee.hpp
class Employee {
public:
 Employee(/*...*/);
 ~Employee();
 static int getCompteur();
private:
 static int compteur_;
```

Déclaration

```
};
```

```
// Dans Employee.cpp
int Employee::compteur_ = 0; // Définition
Employee::Employee(/*...*/) { compteur_++; /*...*/}
Employee::~Employee() { compteur_--; /*...*/}
int Employee::getCompteur() { return compteur_; }
```

Incrémenter sur construction

Décrémenter sur destruction

# Méthode statique – exemple (suite)

- Compteur d'instances

```
// Dans main.cpp
int main()
{
 cout << Employee::getCompteur() << endl;
 auto ptr1 = make_unique<Employee>();
 cout << Employee::getCompteur() << endl;
 auto ptr2 = make_unique<Employee>();
 cout << Employee::getCompteur() << endl;
 ptr1 = nullptr; Mettre à nullptr un unique_ptr désalloue
 cout << Employee::getCompteur() << endl;
 ptr2 = nullptr;
 cout << Employee::getCompteur() << endl;
}
```



Affiche quoi?

## Namespace (espace de noms)

namespace, std, using namespace / using

# Problématique

- Un programme peut utiliser plusieurs bibliothèques développées indépendamment
  - Chacune utilise beaucoup d'identificateurs pour les classes, fonctions globales et parfois objets globaux
- Comment réduire au minimum les conflits de noms?
  - Avoir des longs noms

```
class MaBibliotheque_MaClasse { ... };
```
  - Avoir des espaces de noms Généralement préférable

```
namespace MaBibliotheque {
 class MaClasse { ... };
}
```

# Namespace « std »

- Toute la bibliothèque standard C++ est dans std (ou sous-espaces de noms)

```
#include <string>
std::string texte = "Bonjour";
```

Doit indiquer l'espace de nom (ou avoir un **using ...**)

- Ce qui vient du C en C++ est aussi dans l'espace global

```
#include <cmath> Ce qui vient du C en C++ commence par « c »
double x = acos(0); Peut y accéder avec ou sans std::
double y = std::acos(0);
```

- Les entêtes réellement C sont dans aucun namespace

```
#include <math.h> Nom de l'entête en C
double x = acos(0);
// double y = std::acos(0); //NON
```

## « using » et « using namespace »

- Pour ne pas avoir à toujours indiquer `namespace::`
- **using namespace** importe tous les noms dans l'espace actuel

```
void maFonction() {
 using namespace std; }
```

Localement à la fonction

```
namespace monNamespace {
 using namespace std; }
```

Dans un autre **namespace**

```
using namespace std;
```

Global; jamais dans un .hpp, ça forcerait tout .cpp qui veut l'inclure d'éviter les conflits de noms

- **using** permet de choisir ce qui est importé

```
using std::cout, std::endl;
```

Utilisable aux mêmes endroits que **using namespace**

# Exemple de syntaxe de namespaces

```
namespace espaceA {
int x = 1, y = 2;
}
```

```
namespace espaceB {
int x = 3, y = 4;
}
```

```
namespace espaceC {
int x = 5, y = 6;
struct S { int x; };
S f() { return { y }; }
}
```

Accède en priorité dans l'espace de nom, s'il n'y en a pas il cherche dans l'espace englobant...

```
int main() {
using std::cout;
cout << espaceA::x << ' ' << espaceB::y << ' ';
espaceC::S s1 = { 7 };
//cout << x; _____ Non déclaré
using namespace espaceC;
cout << x << ' ' << espaceB::x << ' ' << f().x << ' ';
S s2 = { 8 }; _____ Utilise celles d'espaceC
using namespace espaceB;
//cout << y; _____ Ambigu
cout << espaceB::y << ' ';
using espaceB::y;
cout << y; _____ using individuel a priorité
//using espaceC::y; _____ Déclaration multiple
```

On peut mettre tout dans un espace de nom: objets globaux, classes, fonctions, ...

Les macros (#define) ne suivent pas les espaces de noms